Tools and Techniques for Building and Testing Software Systems

Glenn S. Fowler
James E. Humelsine
Carl H. Olson

A major issue for each software project is its choice of software building and test processes and the models, methodologies, and tools used to implement them. The project needs efficient and effective procedures to manage change, control versions, and integrate the software to ensure cost-efficient, on-time deployment of a software system. Within AT&T's software-development community, increasing attention is being focused on quicker delivery of higher quality software. As a result, past and current practices are being evaluated and upgraded. This paper outlines some processes, tools, and techniques that offer a potential set of solutions for a variety of software-development platforms. We focus on the SIMP process, which extends some existing tools to provide a software-development process and technology for building, integrating, and testing software systems in AT&T. We believe that SIMP (which stands for *software integration, management, and production*) offers a generic solution in a set of tools configured to meet the needs of many software-development environments.

Introduction

One key to a software project's success is the process used to build (i.e., compile and link), integrate, and manage software. Established procedures and careful management can minimize the problems that often occur when the work of several developers is integrated into a product.

Also, software systems rarely exist in a single version throughout their lifetimes. A system's software does change frequently during development. But even after a system has been delivered to its initial customer, the system will continue to evolve. Several versions of the system often must exist simultaneously to accommodate enhancements (which some customers may elect not to purchase) and to provide different feature sets for different customers.

Without efficient and effective procedures in place to manage change, control versions, and integrate the software, integration problems can and will increase costs and delay deployment of a software system.

Within AT&T's software-development community, increasing attention is being focused on quicker delivery of higher quality software. Because of this added concern, past and current software-development practices are being evaluated and upgraded. Clearly, many practices may need to evolve and use some new or revised processes and techniques to build software, test the software, and provide support tools.

This paper discusses one such process and technique—a software-development process and technology for building, integrating, and testing software systems in AT&T. The process includes version and feature management, viewpathing, building, maintenance, and test management of the different components of a software system. All play important roles in managing the configuration of a software system.

Version and feature management provides order and control of versions and features. It allows software developers and project managers to track when and what features are added to the software and what changes are needed or have been made. In AT&T, modification requests (MRs) are used to identify, document, and track changes to a system's software. Each source-code change must be associated with an MR. (Panel 1

Panel 1. Abbreviations, Acronyms, and Terms

3DFS — three-dimensional file system; enables a user to produce a virtual view of several physically different directory structures, as if they were the union of one directory structure at the file-system level

core — the code used by all parts of the software system, such as a database manager, error handler, or transaction processor

`coshell` — a coprocessing shell; a process that distributes actions to shells on lightly loaded host computers in the local network

evolutionary prototype model — a software model where the product is designed, coded, and delivered over several phases. The project evolves through its different deliverables.

`ksh` — the Korn shell

`make` — a tool that software developers use to compile and link code. It uses explicit Makefiles and the time stamps of source and object code to determine what code to compile and link.

Makefile — a file that describes what is to be compiled and linked. It identifies the source-code files and the executable files to be created by a software build. Software developers use this medium (and makerules) to tell `nmake` what to compile and link.

makerules — a project-specific file that tells how to apply the instructions listed in the Makefiles. It also allows a project to tailor the look of `nmake` to the project's needs.

MR — modification request; the mechanism used in AT&T to identify, document, and track changes to a system's software. An MR is associated with each source-code change and may represent the correction for a problem, an enhancement, or a feature to be added or removed.

`nmake` — new `make`; allows a project to customize its compile-and-link environment and save both state-table and time-stamp information about what is being built

PQRS — Power Quality Resource System

`sh` — the standard UNIX system shell

SBCS — source and binary control system

SCCS — source-code control system

SIMP — software integration, management, and production; a process that offers a set of tools configured to meet the needs of many software-development environments

spiral model — a software model where the product is defined, developed, and delivered incrementally until the finished product is done

throwaway prototype model — a software model where many prototype versions of the code are created, tested, and discarded before the final version of the code is built

time stamp — the date and time that a file was created or changed

TMAS — Transport Maintenance and Administration System

TOPAS — Trunk Operations Provisioning Administration System

TWB — a software-testing system based on `nmake` that implements test cases as Makefiles

viewpath — a list of physically separate UNIX system directories that are viewed as one UNIX system directory. It allows many developers to share the source code and object code, yet gives each developer the sense that he or she has a private copy.

viewpathing — a technique that enables developers to access the files that can be integrated to produce a particular version of the source code

waterfall model — a software model where the phases of the project are divided into individual teams, each totally responsible for its piece of the project. The work passes from one team to another, as in an assembly line.

`xksh` — the Korn shell for an X Window System™ environment. (X Window System is a trademark of Massachusetts Institute of Technology.)

defines acronyms and terms used in this paper.)

Because most software systems today consist of the *core* (i.e., software that is common to all or most parts of the product) and application-specific software, the source code for a given product resides in many files. A *viewpath* tells where to find the files for the system's source code, and the viewpathing technique enables developers to access the files that represent a particular version of the source code. *Building* refers to the process of accessing these files and compiling and linking them

to produce a version of the object code.

Maintenance is the act of correcting errors in code, or updating code to meet changing needs. It does not refer to adding new functionality or features.

Test management refers to the methodologies and mechanisms that control the development and execution of the tests that exercise the associated software objects that represent the product.

In this paper, we describe several UNIX® system tools that, when used together, can provide an efficient means to build and test software (UNIX is a registered trademark of UNIX System Laboratories, Inc.):

- `nmake` — controls the software-build process, so only those files that need to change or be created are built.[1] By using a set of dependencies that are defined by the software developer or programmer, `nmake` decides what code should be compiled and linked and then instructs the computer to compile and link only that code.
- 3DFS — supports a three-dimensional file system structure.[2] With this tool, a user can produce a virtual view of several physically different directory structures, as if they were the union of one directory structure at the file-system level. This is analogous to symbolic links of files. In 3DFS, entire directory structures may be linked, and each user can define his or her own sets of links. When projects use an incremental building process that distributes the product over several directory structures, 3DFS permits these directories to be merged into a single virtual directory structure. Thus, the product appears to be in one directory structure. This is particularly useful when the tools and procedures the project uses expect the product to be in a single directory structure and not distributed over several directory structures.
- `coshell`— coprocessing shell, a user interface for the software-build process. The term *shell* refers to a user interface to a command-driven computer. A shell consists of the set of commands that a user may specify as written, typed, or spoken instructions, or may choose on a menu or from icons or graphical displays.

These tools—`nmake`, 3DFS, and `coshell`—are embodied in the SIMP process, a proprietary tool that is used to integrate and control software builds. SIMP is part of the software-development-environment platform of the Project MOSAIC described by Stacey Gelman, Fred Lax, and Joe Maranzano elsewhere in this issue.[3]

Software Development Models

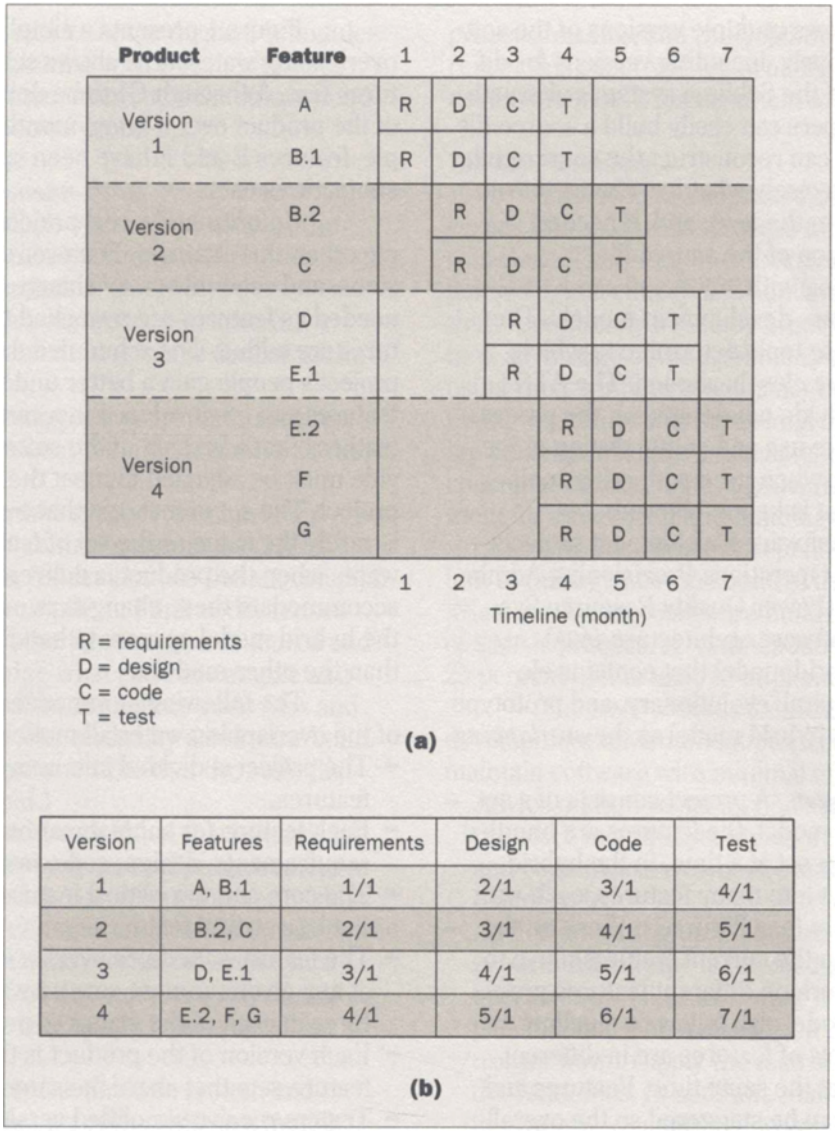
Developing software products is a complex task. Models have been developed and studied to try to understand the tasks involved. The quantitative information in the model is also used to give better predictions of the time and money needed to produce the product.

Several types of models have been and are being used in the industry:

- The *waterfall model* separates the software process into different phases; e.g., requirements, design, development, test, and delivery.[4] The project is completed in each phase and undergoes some degree of final inspection before it can be passed to the next phase. Therefore, problems can cause the project to be rejected and returned to the submitting phase. First used in 1970, this model was a good start and parts of it are still used. On large, complex projects, the delivery date can slip repeatedly. Also, the model does not adapt well to new requests from the customer, and time is compressed for the people in the phases at the end of the "waterfall."
- The *spiral model* separates the work into sets of sequential phases—requirements, design, development, and test—that repeat until the project is completed.[6] The product is developed a piece at a time, until the entire product is ready to be shipped. This give a project a better chance of meeting the customer's time requirements, but schedules can also slip with this model.
- The *evolutionary prototype model* is similar to the spiral model in concept; but after each major set of phases, the product may be delivered to the customer in evolving versions.[6]
- The *throwaway prototype model* can be used with the other models. Prototypes of the product are produced at the requirements, design, and, possibly, development phases.[7] These prototypes are used to validate the phase and show the functionality and feasibility of the product. Usually, the prototypes are replaced by the final versions.

The waterfall model is almost an industry standard, although the spiral model is close behind. Many projects probably use different features from different models.

Because the older models (e.g., the waterfall model) no longer fit the current environment or processes, recent research in software development is moving away from them and toward the newer models (i.e.,



Version	Features	Requirements	Design	Code	Test
1	A, B.1	1/1	2/1	3/1	4/1
2	B.2, C	2/1	3/1	4/1	5/1
3	D, E.1	3/1	4/1	5/1	6/1
4	E.2, F, G	4/1	5/1	6/1	7/1

(b)

Figure 1. In the overlapping waterfall model, the project is divided into many features and subfeatures. Here, four versions of the software product are to be built over a seven-month period, and features B and E have been split into two deliverable subfeatures. Each feature and subfeature has a schedule for requirements, design, code, manufacturing, and test. (a) The overlapping is immediately visible in the time-line schedule. (b) The table schedule assigns completion dates for each development phase.

the spiral, evolutionary prototype, and throwaway prototype models). The research effort is directed at finding a better model to reduce the time and cost of the product and improve the product's quality.

Although they looked promising in theory, these newer models provide only a high-level overview of a project's schedules. This means that the models give little or no detail about the configuration of the software system, information that is critical for managing a software project. Therefore, implementation of these models can be challenging and requires a detailed understanding of configuration management for software.

Many current configuration-management techniques are based on the waterfall model. Therefore, many of them may need to be abandoned, and a new set of techniques introduced to take their place.

The new techniques should be based on new procedures and not necessarily on existing or new tools. Two software-building tools commonly used in AT&T today are the AT&T Sablime™ software product administration system (formerly known as Sable) and `nmake`.[1] The Sablime system keeps track of all changes to a

software system and allows multiple versions of the software to exist simultaneously, including versions for different applications. With the Sablime system's viewpath facility, software developers can easily build a source file for an application. They can reconstruct the source code for any version; track precisely what has changed from one version of the code to the next; and, if needed, restore a particular version of the source file.

These tools, along with 3DFS[2] and `coshell`, can support many software-development models. The procedures that use these tools determine to which model the project is more closely aligned. The AT&T tools cited here generally do not depend on the process or the model. Instead, the use and configuration of the tools—by themselves or when used with other tools—determine the model that is being followed.

Several AT&T software-development projects—including TOPAS (Trunk Operations Provisioning Administration System), PQRS (Power Quality Resource System), and the adjunct software architecture (ASA) project—are using a hybrid model that contains elements of the waterfall, spiral, evolutionary, and prototype models. We refer to this hybrid model as the *overlapping waterfall model.*

Overlapping Waterfall. A project consists of a set of features. In the spiral model, the features are bundled into sets and completed a set at a time. In the hybrid model, the project is split into many features, each with its own waterfall schedule (see Figure 1). Work on the next features starts when the current features move to the next phase. Thus, work on different features proceeds in parallel. The model also follows a pipeline approach; i.e., several sets of features are in different phases of development at the same time. Features and schedules that overlap can be staggered so the overall schedule functions the same as in the spiral model.

Sometimes, a single feature may be too large to fit into one phased delivery. If so, the feature may be split further into subfeatures, each assigned to a different development phase. (These phases may or may not overlap.) Thus, in the hybrid model, each subfeature has its own waterfall schedule. These subfeature schedules can be used to function the same as in the evolutionary and prototype models. Because the feature as a whole is being developed and, possibly, delivered as different subfeatures, the hybrid model follows the evolutionary model's concept.

Figure 1 presents a simplified example of the overlapping waterfall. It shows schedules for seven features (i.e., A through G) to be delivered in four versions of the product over a seven-month period. In this example, features B and E have been split into two deliverable subfeatures each.

Unfortunately, real projects are much more complex than this example. Features may slip into later versions, and schedules may change. Revisions will be needed as features are reworked to correct errors or features are added. The scheduled dates will change as the project's people gain a better understanding of how many features can be developed in a certain period. Often, the features that a version of the software system is to provide must be adjusted to meet the dynamic needs of the project. The set of features that a customer wants initially is rarely the same as the set of feature that the customer wants when the product is delivered. A model needs to accommodate these changes as much as possible, and the hybrid model appears to handle such changes better than the other models.

The following summarizes the main concepts of the overlapping waterfall model:

- The project is divided into many features and subfeatures.
- Each feature (or subfeature) has a schedule for requirements, design, code, manufacturing, and test.
- The core and the critical features are scheduled before the less critical features.
- The feature schedules overlap in real time. Therefore, at any given moment, one may find some of the features in each of the stages of production.
- Each version of the product is the collection of those feature sets that share the same delivery schedule.
- Testers receive simplified versions of the product early in the life cycle and more complete versions later. (*Life cycle* refers to the life of a product or service. It starts with the initial concept for the product or service, and ends with extension or replacement by a newer product or service.)
- An individual feature may go through several subfeature schedules. Each subfeature schedule adds more functionality and complexity to the feature as a whole.

Because requirements, design, coding, manufacturing, and testing schedules are overlapped among the feature set, the parallel nature of this model permits quicker delivery of the product to the customer. The

features, requirements, and, to some degree, schedules are flexible from the beginning with this model. This gives a project room to maneuver and schedule around new features and unexpected problems.

Need for a Cohesive Process

Besides the Sablime system and `nmake`, the UNIX system tools commonly used in AT&T to implement software models include:

- CMTS (configuration-management test system), a configuration-management tool that is similar to the Sablime system.
- CMS (configuration-management system), another configuration-management tool that is similar to the Sablime system.
- SCCS (source-code control system), the basis for nearly all source-code control systems in AT&T. Unlike the other tools, it does not use the MR concept.
- `make`, a tool used to compile and link code. It uses explicit Makefiles and the time stamps of source and object code to determine what code to compile and link. (A *Makefile* identifies the source-code files and the executable files to be created by a software build. A *time stamp* is the date and time that the code file was created or changed.)
- `build`, a software construction tool that extends the `make` command by allowing several programmers to share the same copy of the complete software, while permitting individual changes and testing for each programmer. It uses a viewpath (or an ordered list of nodes that have identical directory structures) to resolve all relative file references when compiling a software object.

Each tool (but particularly the Sablime system and `nmake`) handles only one part of the software model, even though each fills its own niche well. However, the Sablime system and `nmake` are still two tools that do not give their user a unified view or process, which is what he or she needs to implement any software model well.

Many AT&T organizations have integrated these tools into software-development environments. SIMP, a product of AT&T's Software Technology Center, is such an environment. It consists of a small set of commands and a set of product-building processes. It integrates `nmake` and the Sablime system into a tool that simplifies the tasks needed to build a system's software.

The authors' areas of expertise are the SIMP process, `nmake`, and the Sablime system. Therefore, the rest of this paper will focus on these three software development and management tools.

The SIMP Process

The need to integrate existing tools and processes into a cohesive process for software development was clear. Therefore, the SIMP process was developed using original concepts in addition to proven concepts that were already in use by other software projects. We took advantage of the Sablime system and `nmake` by using their configuration-management features in a functional way rather than by reimplementing them. By reusing existing tools and processes in combination with new concepts, we were able to provide a cohesive environment for developing and building software systems.

The SIMP process depends on the Sablime system to manage MRs and control the source code, and on `nmake` to manage software builds. However, 75 percent of SIMP is process, recommendations, and advice, while 25 percent of it is tools to implement the process.

The SIMP process focuses on how to set up a development environment properly and then build and maintain software with minimal effort. This includes:

- Advice on how to set up node structures. The *node* is the place in a file system where source code resides and code files are compiled and linked to produce the object code.
- Advice on setting up Makefiles and makerules for `nmake`. A Makefile is the medium a software developer uses to tell `nmake` what he or she wants to compile and link. The *makerules*, a project-specific file, tell `nmake` how to apply the instructions that are listed in the Makefiles. In addition, makerules allow a project to give `nmake` a tailored look, as though it were customized for that project.
- Help for setting up development schedules, such as the one in Figure 1.
- Advice on how to set up viewpaths for software builds and run-time environments. *Software build* refers to the act of accessing a version of the source code from the source-code control system and, then. compiling and linking the code. However, this version may be only a subset of the entire product. Viewpathing will pick up the remaining pieces of the product. *Run-time environment* refers to the new version of the product that is constructed by the viewpath. We want to be

able to test this new version. The viewpath used to construct the new version can also be used to define and declare a run-time or testing environment for it.
- Tools to help developers and project builders automatically extract the source code using the Sablime system and build the code using `nmake`. The Sablime system keeps track of all MRs and the different versions of code associated with each one. Thus, a user can construct a version of the files that a given set of MRs affected. The Sablime system passes these new "version" files to `nmake`, which checks their time stamps and other dependencies. From this, `nmake` decides what files should be compiled and linked, how, and in what order.

Although the SIMP process was designed with the overlapping waterfall model in mind, the process can be adapted to other software models.

One philosophical goal of SIMP is to ensure that software projects initially define a flexible development environment that meets their current needs yet allows future growth. To avoid environment redesign, the development environment should focus on preventing problems, rather than on reacting to them.

Obstacles and Solutions

A major obstacle for software manufacturing has been that a project must try repeatedly to build the software before it successfully creates a version of its product. Multiple attempts are necessary when a software build fails because of compilation errors such as:

- Syntax error in source code
- Failure to find the header files
- Failure to find the library files
- Failure to find definitions for variables, functions, and symbolic constants.

Rarely is only one compilation error encountered. Often, many retries are needed to find and correct all the errors. This makes it difficult to implement any software development model, especially the overlapping waterfall model with its asynchronous schedules.

Dissimilar Environments. Why do so many compilation errors occur during integration builds? Through the experience we gained from developing the SIMP product, we were able to find the major cause of the problem: A developer's local environment often contains customized libraries and header files that are not in the integration environment. As a result, software that compiles without error in the developer's local environment may produce compilation errors in the integration environment.

Each developer who submits code to a software build compounds the problem. Thus, if a project has several developers, the probability increases that any integration build contains errors and delivery of the integrated software to testing will be delayed.

Clean nodes for builds. One way to remove this type of problem is for developers to use a new node—not their local, customized work space—to build their code. Then, because a developer's builds occur in a standardized environment instead of a local work space, most compilation errors will be found before the software is submitted to an integration build.

Although the Sablime system and `nmake` allow a software developer to use a clean node for each build, the procedures are complicated and time consuming. Therefore, few developers take the time and effort to use clean nodes. With the SIMP process, building a clean node is much easier because a SIMP command automates the Sablime and `nmake` procedures.

More-frequent integration builds. Another way to reduce the number of build attempts needed to create a single version of the product is to schedule more project-integration builds. Although this may seem contradictory, the increased frequency of the project-integration builds helps to reduce deadline anxiety. The developers are more likely to deliver better quality code for a given build, so fewer and smaller changes are needed for subsequent builds. Why should this be true?

When a developer works on a project that has infrequent integration builds, he or she feels pressured to meet the current build's deadline even if the code is not ready. Because of deadline pressure, developers (and their managers) assume incorrectly that it is better to supply code of poor quality to meet a deadline than to wait until the next build to provide good-quality code. If integration builds are done more frequently, a developer does not feel the same deadline pressure and will wait to submit code of good quality.

In addition, more-frequent integration builds decrease the amount of code that changes in each build. Fewer changes reduce the risk of a compilation error. Also, when an error does occur, the area potentially affected tends to be smaller.

This scheduling technique has been used successfully on several projects including TOPAS and TMAS

(Transport Maintenance and Administration System).

Other Obstacles. The following obstacles can also contribute to problems with the software-development environment:

- Software projects do not use the viewpath concept. If so, the project must make concessions in order to develop the code. For example, on small projects, each developer might have a complete copy of the project in his or her file system. The copies will use up large amounts of storage, and individual versions will become out of date compared to the official version. Larger projects might have one communal node structure, where each developer places the latest version of his or her software to compile. However, developers run the risk of overwriting the code and may not keep their source-code control system up to date.
- Developers have an explicitly coded VPATH entry in their UNIX system `.profile`, and do not use a project-controlled table-lookup function. The VPATH variable specifies the viewpath and should be accessed through a look-up table that the project implements and controls. (A `.profile` is a script, often user designed, that is executed automatically during the login process to establish the user's local environment for that session.)
- Developers are responsible for setting up their own environment variables for software compilation and run-time testing. If so, they may be using old or different support software. Also, it would be more difficult to distribute updated support software (e.g., a new compiler) to each developer or to add a new developer to the project. Instead, the developers should use a project-controlled environment-declaration program, so that they all use the same support software and the same unit-test environment.
- Some developers have defined a local environment that is inconsistent with the environment used by the other developers and the project as a whole. A developer who does this may use old or improper versions of the support software, such as compilers, interpreters, and database managers. As a result, the code might work fine in the developer's unit-test environment but exhibit strange behavior in the integrated system-test environment. All developers should be using the project-controlled environment.
- The project has only one installation and test area on the system, which means only one version of the product can be tested (usually the integration-test version). Instead, the project should have many installation and test areas, one for each "testable" version of the product. The multiple areas allow each developer who has testable source code to have access to all important versions of the product, particularly if a customer's problem needs to be examined.
- Different environment structures are used for development and for installation and production. This means the software must be shifted to another environment structure, which adds an extra step and provides another opportunity for error. Also, developers do not have an environment that allows them to develop software as it will appear to customers. To avoid these problems, the development environment should look as much like the final product's installation environment as possible.
- MRs for project builds are gathered manually, e.g., as saved electronic-mail messages, as telephone messages, or on slips of paper. Messages might be lost or forgotten. Also, when people finally type the MR forms, the typing may be inaccurate, i.e., the typist may include incorrect MRs or lose important ones. One person could hold up the software build by asking for just "five more minutes." Instead, MRs should be gathered in a controlled way, preferably electronically, as part of the build process.
- Previous modifications to the product are lost when new versions of the product are shipped. When this happens, it is not easy to find or fix problems in the older version of the code. The project should implement a source-code control system and use it properly.
- The versions of the product shipped to customers are not the version that is stored in the source-code control system. (The danger of this problem is the reality of the next one.) The project should ensure that its source-code control system is used properly.
- Versions of the product that are shipped cannot be reproduced in the laboratory. This problem happens when either or both of the previous problems exist. As stated previously, the project should implement a source-code control system and ensure that it is used.
- The development-node structure is too large or complex to manage and to grow into the new generics. New generics usually mean new features, so the code that implements these features must reside somewhere in the node structure. The project's node

structure should allow new directories and source files to be added easily. That is, it should not require that the entire node structure, makerules, Makefiles, and existing source code be reengineered.
- Development environments change from generic to generic, causing confusion among developers and incompatibilities between generics. Radical change tends to cause chaos and errors for a while. Because change is necessary, the development environment's design should be left open for growth and change, to minimize chaos and errors.

One Solution. The SIMP process and tools mentioned above help to overcome these problems and obstacles. When a project uses the SIMP process and these tools, developers have the means to check the compilation quality of their code; and the project can schedule its integration builds as frequently as needed. SIMP also provides recommendations and support to help resolve all the obstacles mentioned in this section.

Project-Management Planning

Over a product's lifetime, many versions of the software will be built and new deliverable versions will be produced frequently. The project managers are responsible for scheduling delivery of each version and mapping its functionality, as illustrated in Figure 1. However, both the schedule and the functionality are subject to change as the product advances in its life cycle.

Beside defining the schedules, project managers must also monitor progress and take action when needed. The processes and models do not run or manage themselves. If left unattended, they will atrophy as all natural systems do. Project management is responsible for defining the processes to be used and, then, monitors the processes and adjusts them as needed. In addition, it ensures that the day-to-day work continues.

The SIMP process provides recommendations to project management on:
- How to organize schedules.
- How to divide the work into features and delivery loads.
- How to define node structures.
- How to schedule software builds.
- How to organize project-management meetings and manage the dynamics of the meeting.
- What reports to generate.
- What to monitor.

Clearly, a project incurs costs when it uses the overlapping waterfall model and the SIMP process. If integration builds occur more frequently, then more time is needed for additional compilations for each build. Also, more memory space will be needed to store the additional versions of the product. In addition, these multiple versions put greater strain on the test team, because more versions are being built and delivered to testing. The sections that follow discuss processes and tools that reduce the time, space, and testing resources needed.

Test-Environment Challenges

Today's software-testing practices are labor intensive and require that test organizations expend many human hours to develop, execute, evaluate, and manage tests and test results. This levies a heavy expense on the test organizations.

A wide array of methodologies and concentrations exist for software verification, e.g., unit, integration, system, regression, load, and stress testing. Throughout AT&T, the test organizations use a wide variety of tools and methodologies that each have strengths and shortcomings. The consistent weaknesses in the current test processes are:
- Independent and unique test systems have steep learning curves. Thus, each requires a substantial investment in start-up time.
- The test system does not offer version control for test cases. A test case is a project-defined granularity, and consists of the particular suite of tests associated with a version of the software product to be tested. Without version control, the test system might use an inappropriate test case to test a particular version of the software product.
- Testing must be integrated manually with other processes in the product life cycle, including:
  — Systems engineering, the stage in which the requirement is defined and the architecture and functionality are specified
  — Project management
  — Development
  — Manufacturing.
- Test development and test execution are time consuming.
- An experienced administrator is usually needed for the test system.

The general strengths of some existing systems are:
- The test formats and the definitions of important data fields are standard.
- The test-organization strategies are well structured

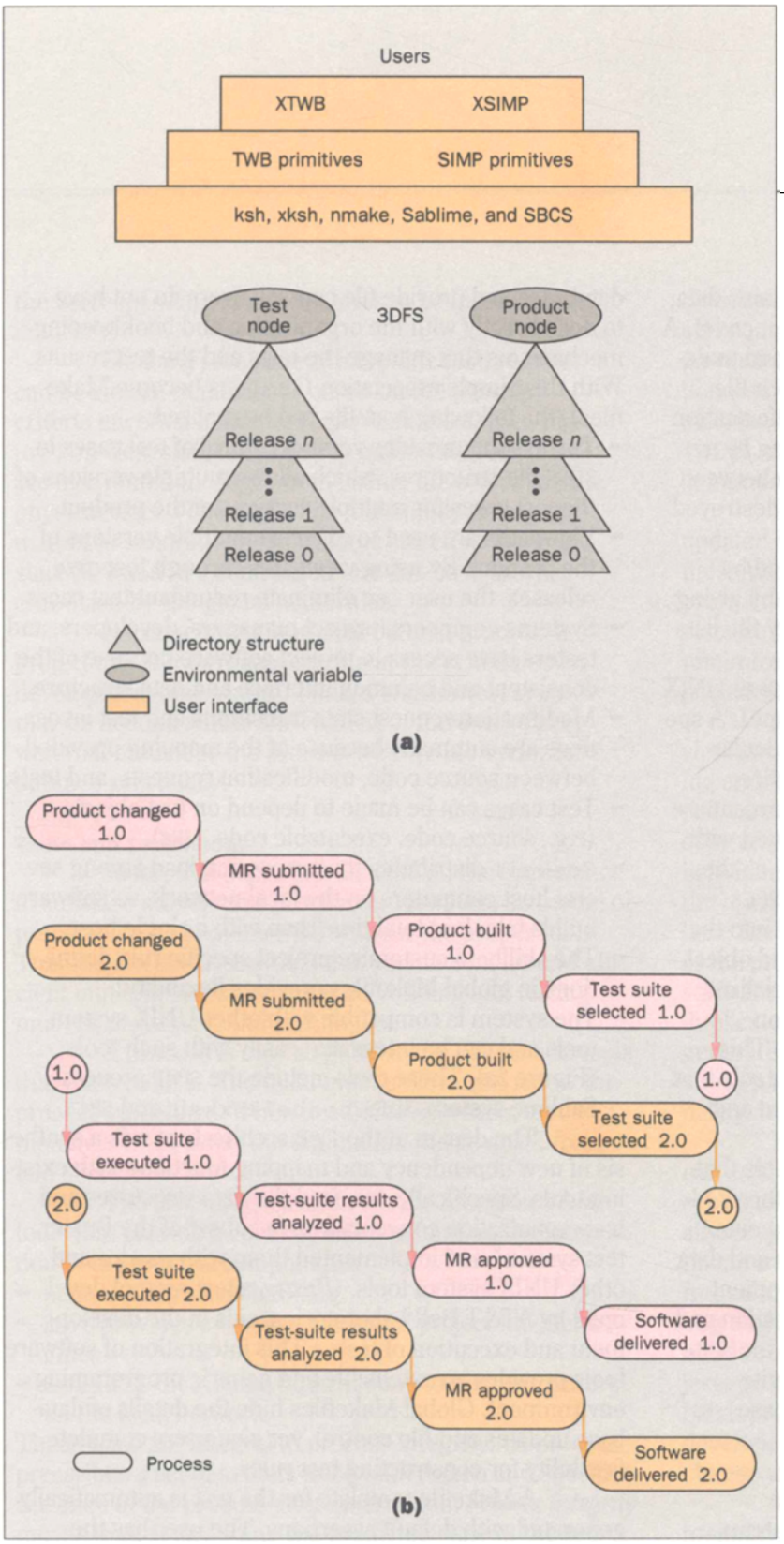


Figure 2. A unified system allows the same management tools to be used for product development and test. (a) In this integrated architecture, the SIMP process manages software development and Tester's Workbench (TWB) manages and provides the test environment. Both use `nmake`, the Sablime software product administration system, and SBCS to manage and control the source code and work with coprocessing shells, including `ksh` and `xksh`. Also, 3DFS "links" files in several physically different directory structures, so a user sees a single virtual directory structure at the file-system level. (b) Overlapping waterfall with SIMP and TWB. Each source-code change is identified in a modification request (MR). SIMP depends on the Sablime system to manage MRs and control the source code, and on `nmake` to manage software builds. An MR's state can be altered manually or based on the pass/fail criteria encoded in the test-case Makefiles.

and use relational selection and extraction mechanisms for test cases.

Commonality in Process. The technology that already supports development platforms can also be used in the testing process. A test that exercises a software object generally consists of the same basic components as the target product under test. Software development in a UNIX system environment means the software must be organized into an arbitrary number of files and directories; really four file types:

- *Regular file* — any UNIX system file that contains data, text, or program instructions (including object code). A file subsystem controls access to these files and maintains status and control information about each file.
- *Pipe file* — a file that contains unformatted information stored in data blocks for access and formatting by a user program. A pipe file is used to pass data between two processes; hence, the file's contents are destroyed once the file is read.
- *Directory file* — a group of files (usually related by topic or function) and "placed" in a directory by giving the directory a name. (In reality, the directory file lists the files that reside in that directory.)
- *Special file* — any file that has special meaning to UNIX system, e.g., device drivers such as /dev/qtape1. A special file serves as the interface between a particular device and the software that manages the device.

The software products developed in this environment are built with these data structures and are maintained with many proven and reliable UNIX system tools—i.e., the SIMP process, the Sablime system, `nmake`, and SCCS.

A typical suite of tests can be separated into the same basic elements. The C-language source and object code, UNIX system shell scripts, and other specialized test software can be reduced to the least-common-denominator UNIX system files and directories. This allows us to take advantage of the management tools that are being used in AT&T in software development and manufacturing.

In a unified system, the management tools that are used for tracking, building, and organizing the product can also be used to manage, develop, and execute test suites. (See Figure 2a.) Common interfaces and data structures, along with mapping between development and test processes, would allow automatic invocation and state transitions of tests and MRs. That is, when the state of the MRs changes to a particular state, a test suite would be invoked automatically; similarly, test-case results could cause MR states to be updated.

A Test Solution and System Overview. The Advanced Computing Environment group in the Advanced Software Engineering and Expert Systems Department of AT&T Bell Laboratories has developed a testing system that is based on `nmake`. This proprietary system, the Tester's Workbench or TWB, implements test cases as Makefiles. Because base rules and global Makefile definitions are used to update the system's databases and provide file control, users do not have to deal directly with the organization and bookkeeping mechanisms that manage the tests and the test results. With this simple association (i.e., tests become Makefiles), the following benefits can be realized:

- The system provides version control of test cases in 3DFS file structures, which allows multiple versions of the test cases for multiple versions of the product.
- Viewpaths are used to exercise multiple versions of the product. By using viewpaths through test-case releases, the user can eliminate redundant test cases.
- Systems engineers, project managers, developers, and testers have access to project software because of the consistent and common interface and data structure.
- Modification-request state transitions and test invocations are automatic because of the mapping provided between source code, modification requests, and tests.
- Test cases can be made to depend on test objects (e.g., source code, executable code, MRs).
- `Coshell` distributes the processing load among several host computers on the local network, so software builds take less total time than with a single host.
- The ability to customize project-specific rule definitions in global Makefiles provides flexibility.
- The system is compatible with other UNIX system tools and can be integrated easily with such tools (Figure 2a). These tools include the SIMP process, Sablime system, 3DFS, `coshell`, `xksh`, and SBCS.

The design of the TWB architecture was a synthesis of new dependency and mapping functions with existing tools. Specifically, we used the data structures and test-organization concepts from a subset of the Buster test system[8] and implemented them with `nmake` and other UNIX system tools. (*Buster*, a testing tool developed by AT&T Bell Laboratories, aids in the development and execution of tests.) This integration of software tools provides an extensible and generic programming environment. Global Makefiles hide the details of database updates and file control, yet give users complete flexibility for constructing test rules.

A Makefile template for the test is automatically generated with default assertions. The user has the option to program the targets, prerequisites, and associated actions. Suites of tests are selected relationally and run on the software-product targets. Test extraction may be explicit (e.g., the suites are specified) or implicit (e.g., certain source-file changes and MR state changes cause

the corresponding test-case suites to be invoked automatically).

Similarly, the state of a modification request can be altered manually or based on the pass/fail criteria encoded in the test-case Makefiles. (See Figure 2b.) Test status is stored in makestate files that are distributed throughout the same file system as the physical tests. (The *makestate files* supply timing formation to `nmake`.) The test status stored in the makestate files and in a centralized test database is synchronized by the global makerules.

Because TWB automates the test process, a project can allow greater overlap in the schedules for development and testing. Thus, a shorter test cycle may be needed within the context of the overlapping waterfall model for the product's entire life cycle, as outlined previously.

Tools and Technology

So far, this paper has focused on the importance of software models and processes. However, no model or process can be implemented without the proper tools. This section discusses tools that permit concise and efficient implementation of software building and testing models, processes, and techniques.

A process is only as good as the people and tools that implement it. The team must believe and follow the process guidelines. When deadlines approach, team members must avoid the temptation to use quick fixes and shortcuts.

The SIMP process and TWB are based on a few tools that provide a concise, efficient, and unified abstraction of the process components:

- `nmake` controls all building.
- 3DFS provides a consistent view of project file hierarchies.
- `coshell` distributes build actions to host computers on the local network.

These tools are integral to process integrity. (A process prescribes a set of actions to be undertaken and specifies the state of the process after each action. *Process integrity* means that, at any point, the predicted state in the process model matches the real state of the process.)

Other models delay software builds because of computation costs, but the SIMP process encourages frequent builds. It is able to do this because `nmake` and `coshell` diminish the costs of the software builds by distributing the builds. However, the SIMP and TWB models require more storage space and more computing power from the computing environment than the traditional software building and testing approaches. In particular, the flexibility that developers enjoy with SIMP and TWB is realized only with adequate file-system space and computing cycles.

The project areas that all developers share are updated frequently to maximize sharing of the generated files. Also, before a developer submits changes into the project areas for integration, he or she must do a complete build in a clean, standardized environment. This minimizes integration errors that might be caused by the peculiarities of his or her private environment.

As Figure 3 illustrates, 3DFS provides a unified view for all project files. It joins the concepts of file sharing and file partitioning, making it a natural for managing software releases. While developers may share a single copy of the process files, 3DFS also efficiently supports an individual's code changes that do not affect the rest of the team's code. Also, because all SIMP and TWB tools have the same view of the shared files, a separate copy of the project source files is not needed for each clean software build. Although source-file sharing eliminates duplicate source-file copies, each developer build could generate copies of all the project target files. Therefore, enough temporary file space is still needed to accommodate private software builds.

In the model that SIMP produces, software builds can easily become the bottleneck. Insufficient computing cycles can bring all project work to a standstill. Traditionally, projects have attacked the problem by migrating to mainframe computers for project software builds. Build frequency usually slows as the mainframe computers bog down, and all the negative effects of infrequent builds (which SIMP addresses) take control. Some projects go to the extreme of submitting software builds as batch processes for low-priority execution. As a result, a developer may not see the results of a one-line code change until the next day.

The recent trend has been away from a few monolithic mainframe computers that all developers share, and toward networked desktop microcomputers or workstations, one for each developer. The networked machines provide a solution to the compute-cycle crunch. Even in the busiest organizations, the local network usually has a few idle workstations. These idle cycles

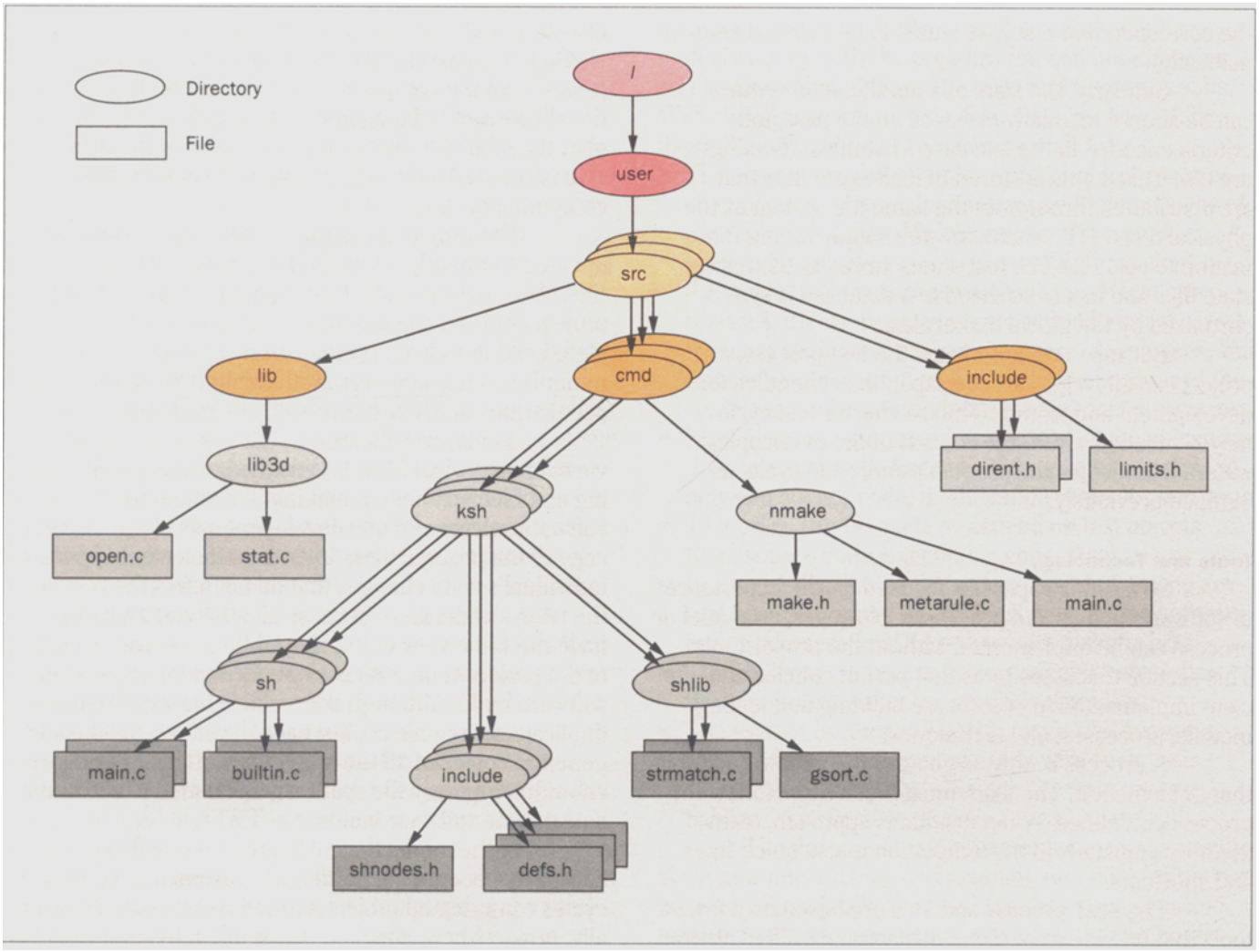


Figure 3. When files are distributed over several directory structures, 3DFS allows them to be merged into a single virtual directory structure. This is analogous to symbolic links of files. This three-dimensional view of a typical file system allows a user to trace (or construct) the path to any directory, subdirectory, or file.

can be used to reduce build-cycle times dramatically.

To distribute builds on the local network, the build tool must first be able to execute actions in parallel. This capability is provided as a fundamental part of `nmake`'s build model. No Makefile changes are required for parallel execution. To achieve parallel execution, `nmake` sends actions over a pipe to a shell coprocess and simply encapsulates each action—i.e., as `{action}&`—for concurrent execution. Action status is returned to `nmake` on another pipe from the shell.

To implement network execution, we replaced the shell coprocess with `coshell`, a process that distributes actions to shells on lightly loaded host computers in the local network. All network specifics are concentrated in `coshell`.

As with parallel execution, no Makefile changes are required to enable network execution. The network-specific partitioning is so complete that, when the prototype `coshell` was first tested, `nmake` was able to provide network execution without changing its implementation (i.e., the `nmake` executable did not change). Local

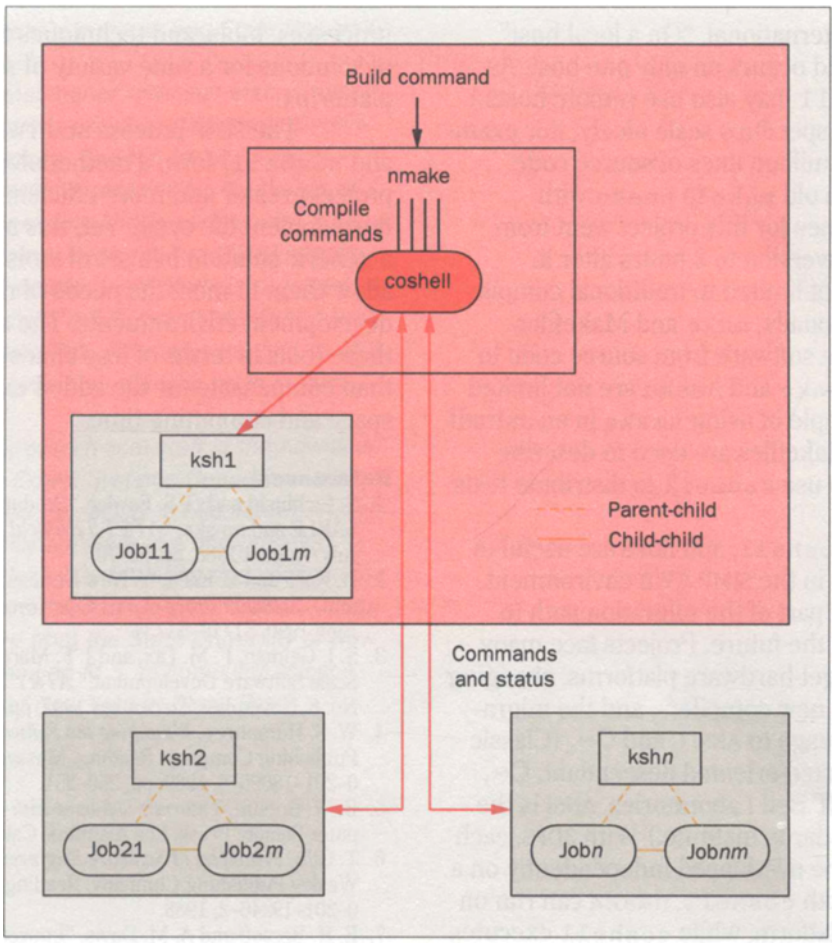


Figure 4. Software builds can be distributed when `coshell` is used with `nmake`. The "frames" represent host computers on a local network. The user is on the local host, where he or she runs `nmake`, which sends actions to `coshell`. `Coshell` sends actions to shells (i.e., `ksh`) that are running on remote hosts (or possibly on the local host). "Job" refers to the child processes (i.e., actions requested) of the parent `ksh`. There may be many children.

versus remote execution is controlled by a single environment variable `COSHELL=`*path* that contains the pathname of the shell coprocess. This variable may be set to the path to the Korn shell, `ksh` (the default coprocess); to the standard UNIX system shell, `sh`; or to `coshell`.

`Coshell` runs as a background process for each user across login sessions. Because it uses the standard `rsh` or `remsh` to connect with `ksh` on remote systems, no special privileges are needed to install, run, or administer `coshell`. ("Standard" means `rsh` is UNIX System V, while `remsh` is BSD—i.e., the version of UNIX system from the University of California at Berkeley. Both remote shells provide the same functionality.)

The only administrative requirement is that all hosts on the local network must share the same filename space. (The collection of all file pathnames forms the filename space.) This is not restrictive, because most local networks are already configured this way for administrative sanity.

The maximum number of concurrent actions is controlled either by the `nmake` option, `-j` *nproc*, or by the environment variable, `NPROC=`*nproc*, where *nproc* is the number of processes. What ultimately limits the speedup that `coshell` can provide over sequential execution is the network file-transfer bandwidth. Build speedups of four to six times have been common across a variety of software-manufacturing environments. Thus, with six idle hosts available and `NPROC=6`, one could expect a build to be completed four to six times faster. To compile and link `nmake` on a Sparc® 1 workstation requires 6 minutes on a local host and only 1 minute with `coshell`. This workstation is rated at about 10 MIPS (million instructions per second), and `nmake` consists of about 15,000 lines of source code. (Sparc is a registered

trademark of SPARC International. "On a local host" means the software build occurs on only one host. As Figure 4 shows, `coshell` may also use remote hosts.)

In practice, the speedups scale nicely. For example, a project that has 1 million lines of source code recently converted from old `make` to `nmake` with `coshell`. The build times for this project went from 10 hours before the conversion to 2 hours after it.

Speedups are not limited to traditional compile-and-link builds. (Traditionally, `make` and Makefiles describe how to compile software from source code to executables, although `make` and `nmake` are not limited to this. TWB is one example of using `nmake` in an untraditional way.) If `nmake` Makefiles are used to describe tests, then TWB can also use `coshell` to distribute tests on the local network.

Also, `nmake`, `coshell`, and 3DFS are useful in their own right, not just in the SIMP-TWB environment. They are a fundamental part of the migration path to the software systems of the future. Projects face many difficulties: multiple target-hardware platforms, changing development platforms, new compilers, and the migration from classic C language to ANSI C and C++. (Classic C language and its objected-oriented descendant, C++, were developed at AT&T Bell Laboratories. ANSI is the American National Standards Institute.) With 3DFS, each step in the change can be overlapped independently on a working system. And with `coshell`, `nmake` can run on a tested development platform, while `coshell` executes target-dependent actions (e.g., compilers) on a different target platform.

A key to the success of these three tools is that they function well together *and* each functions effectively with either of the others. If 3DFS is not available for a particular development environment, then `nmake` provides an equivalent, although not transparent, fallback mechanism. Also, `coshell` is a compatible add-on for networked environments. In short, a correct `nmake` description (i.e., it does not use special 3DFS or `coshell` constructs) will work with any combination of 3DFS and `coshell`.

Conclusion

Every software project must address the issues of how to improve its software building and test processes and what models, methodologies, and tools to use to implement those processes. This paper has outlined processes, tools, and techniques that offer a potential set of solutions for a wide variety of software-development platforms.

The SIMP process and TWB (together with 3DFS and `coshell`) form a methodology that gives a software project greater and more efficient control of the product-development life cycle. Yet, this methodology also offers a generic solution in a set of tools whose configurations allow them to meet the needs of many, heterogeneous development environments. The savings achieved with these tools in terms of less chaos and less rework more than compensate for the added expenses for memory space and computing time.

References

1. 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.
2. D. Korn and E. Krell, "A New Dimension for the UNIX® File System," *Software Practice and Experience*, Vol. 20, Supplement 1, June 1990, S1/19–S1/34.
3. S. J. Gelman, F. M. Lax, and J. F. Maranzano, "Competing in Large-Scale Software Development," *AT&T Technical Journal*, Vol. 71, No. 6, November/December 1992, pp. 2–11.
4. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Company, Reading, Massachusetts, ISBN 0–201–18095–2, 1989, pp. 250–251.
5. B. W. Boehm, *Tutorial: Software Risk Management*, IEEE Computer Society Press, Los Alamitos, California, 1989, pp. 26–37.
6. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley Publishing Company, Reading, Massachusetts, ISBN 0–201–19246–2, 1988.
7. E. H. Bersoff and A. M. Davis, "Impacts of Life Cycle Models on Software Configuration Management," *Communications of the ACM*, Vol. 34, No. 8, August 1991, pp. 105–118.
8. K. C. Archie and R. E. McLear, III, "Environments for Testing Software Systems," *AT&T Technical Journal*, Vol. 69, No. 2, March/April 1990, pp. 65–75.

(Manuscript received July 17, 1992)

Glenn S. Fowler *is a distinguished member of technical staff in the Advanced Software Technology Department with AT&T Bell Laboratories in Murray Hill, New Jersey. He is currently involved with research on configuration management and software portability, and is the author of* `nmake`*, a configurable ANSI C preprocessor library, and the* `coshell` *network-execution server. Mr. Fowler joined the company in 1984. He has a B.S.E.E, an M.S.E.E, and a Ph.D. in electrical engineer-*

ing, all from Virginia Polytechnic Institute and State University in Blacksburg, Virginia.

***James E. Humelsine** is a member of technical staff in the Advanced Software Engineering and Expert Systems Department at AT&T Bell Laboratories' Red Hill facility in Middletown, New Jersey. He currently is responsible for design and development of the SIMP process and is working with a group in AT&T's Software Technology Center to turn SIMP into an internal AT&T product and deploy it. Mr. Humelsine joined the company in 1985. He received a B.S. from The Pennsylvania State University in University Park, Pennsylvania, and an M.S. from the University of Wisconsin at Madison, both in computer science.*

***Carl H. Olson** is a member of technical staff in the Advanced Software Engineering and Expert Systems Department at AT&T Bell Laboratories' Red Hill facility in Middletown, New Jersey. He is responsible for software tool and process development, particularly for system testing and verification applications. Mr. Olson joined the company in 1986. He has a B.S. in computer science from the State University of New York College at Oswego, and an M.S. in computer science from the State University of New York at Stony Brook.*