AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Build—A Software Construction Tool

By V. B. ERICKSON* and J. F. PELLEGRIN*

(Manuscript received August 19, 1983)

The `build` tool is used as a sophisticated method of generating and modifying software systems. `Build` is being used successfully by a number of *UNIX*™ software-based projects at AT&T Bell Laboratories. `Build` is an extension to the `make` program that permits several software developers to independently make a collection of software while sharing the same fully populated set of directories, with the changed files residing in their own directories. An important concept in using `build` is *software view*, which represents the selection of a particular version of software for a generation environment. For example, a developer's view of a software system generally includes all of the current "official" software perturbed by the developer's private modifications to the system. A testing team's view may be the current official software perturbed by changes that a set of developers have made and have submitted for project system testing. A system user's view is a fully tested and released version of the software. The function of `build` is to simplify the administration of the different views of the software system. The `build` tool is being used by a number of large software development projects as the primary software generation tool. `Build` plays a central role in the development strategies and standards used in these projects.

I. INTRODUCTION

The `make` tool that is available with the *UNIX*† operating system is used as an aid in the construction of software. A specification file, called a *makefile*, contains a description of the software targets that

* AT&T Bell Laboratories.

† Trademark of AT&T Bell Laboratories.

are to be built, a list of files that are needed to construct each specific target (the target's dependency list), and the commands to be executed to create the target. A set of directory structures populated with source files and cooperating makefiles may be used for the construction of a larger collection of software, or for the propagation of changes to the software. The `build` tool is an extension to `make` that permits several developers to independently make changes to a collection of software while sharing the same fully populated set of directories, with the changed files residing in their own directories. Through its use, support for the testing of individual developer versions of the system can be provided without the overhead of redundant storage consumption.

A basic need in a project of two or more people is to provide an environment in which they can work independently on modifications to an existing system of software. If several people wish to test their private modifications to the same portion of the system, independent testing can be achieved by replicating the complete set of populated directories for each developer and permitting them to make changes to their own copy. This is workable in a small project, but the space consumption can rapidly become prohibitive. The desire to share a single copy of the complete software among many developers, while still providing for individual developer changes and testing, was the motivation for the `build` tool. The remaining sections in this paper discuss some basic concepts used in `build`, describe its behavior in detail, provide a simple example, and describe the use of `build` in a particular large software development project.

II. BASIC CONCEPTS

2.1 The make tool

The `make` tool automates the software generation steps that come between the editing and testing phases by executing specified commands to reprocess any and all files that have been affected by the editing.[1] This eliminates manual effort required to reprocess files, potential errors caused by forgetting to reprocess files, and overhead of unnecessarily reprocessing files.

The `make` tool executes commands to generate target files as specified by the contents of a makefile description file. A makefile contains a representation of the graph of file dependencies. Each nonleaf file in the dependency graph is a target file, which may have an associated set of regeneration commands; its descendants are the files from which the target file was created. If any of the target file's descendants are modified, the target file must be regenerated to maintain a consistent system.

For example, Fig. 1 depicts a populated *UNIX* operating system

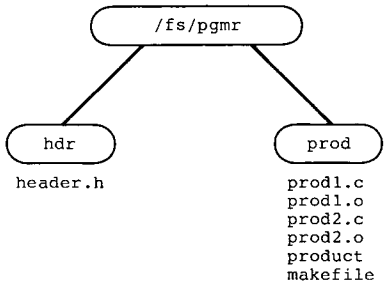

Fig. 1—Fully populated node.

directory structure, including all files necessary to generate the load module `/fs/pgmr/prod/product`. Figure 2 shows the graph of file dependencies for the load module, with file generation commands shown in brackets. The corresponding makefile is shown in Fig. 3. The file names to the left of the colons are the target files, and correspond to the nonleaf files in the graph. The files to the right of the colons are the files that the targets depend on, corresponding to the nonroot files in the graph. The `make` tool examines each file in the graph, checking to see which target files need (re)generating. If a target file does not exist or is dependent on a file that has a later modification date, then the target is (re)generated by executing the generation commands associated with it.

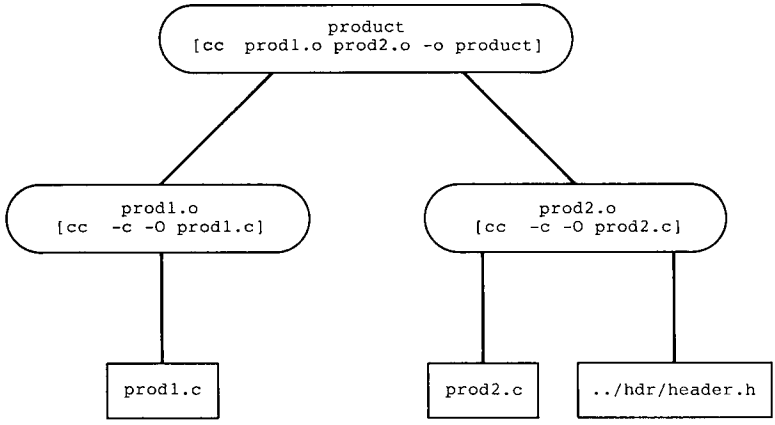

Fig. 2—File dependency.

In the above example, if a developer modified the file `../hdr/header.h` and then executed `make` from within the `/fs/pgmr/prod` directory, `make` would execute the following commands:

```
cc -c -O prod2.c

cc prod1.o prod 2.o -o product
```

The modifications to the file `../hdr/header.h` cause the target, `prod2.o`, to be regenerated. The resulting execution of `cc` causes the file, `prod2.o`, to be updated. This in turn forces the target `product` to be rebuilt.

2.2 Individual software views

Individual software views are versions of the software system that are unique to an individual developer or to a particular set of developers, such as system testers. Different views are formed by combining different collections of files from the system. These collections are stored in separate nodes. A *node* is a set of *UNIX* operating system directories, all of which share the same common ancestor directory, called the *root* directory of the node. A node for a software project is defined as a project-standard set of directories that are sufficient to contain the complete set of project files. Figure 1 is an example of a node, which is populated with all of the files in the system. The root of the node is the directory `/fs/pgmr`. Any file that can be reached from `/fs/pgmr` is contained in the node. A file in a node is identified by the relative file name describing the path from the root of the node to the file. In Fig. 1, the relative file name `hdr/header.h` identifies the file `/fs/pgmr/hdr/header.h`. Multiple instances of a node may be established by duplicating the same set of directories, each below a different root directory. Each instance may contain versions of some or all of the project files within the directories.

Individual software views for developers are established by having each developer work in a separate node containing only those files that the developer needs to change. The developer then accesses all other project files through a separate project-wide shared node that contains a complete set of all the project files.

This combining of individual and shared files in separate nodes to express a particular software view is achieved through the specification of a viewpath. A *viewpath* is an ordered list of nodes, each of which has the same directory structure. The viewpath is used to resolve

```
product:  prod1.o  prod2.o
     cc  prod1.o prod2.o -o product

prod1.o:  prod1.c
     cc  -c -O prod1.c

prod2.o:  prod2.c  ../hdr/header.h
     cc  -c -O prod2.c
```

Fig. 3—Makefile contents.

references to files. A file, identified by its relative path name within the node's directory structure, is located within the viewpath by searching in its directory within each successive node in the viewpath until it is located. Any additional versions of the file in subsequent nodes in the viewpath are ignored. In this way, the viewpath determines which version of each file in the software system is to be used in a particular software view that consists of a set of populated nodes.

To specify a viewpath, which is needed by `build`, users define the viewpath using an environment variable called `VPATH` or an option on the `build` command line. The viewpath specification consists of a list of directories, representing nodes, separated by colons.
In Fig. 4, three nodes are depicted:

`/fs/project`—complete project-released software node

`/fs/pgmr1`—private node of developer named programmer 1

`/fs/pgmr2`—private node of developer named programmmer 2.

Two different viewpaths for separate developers are indicated by

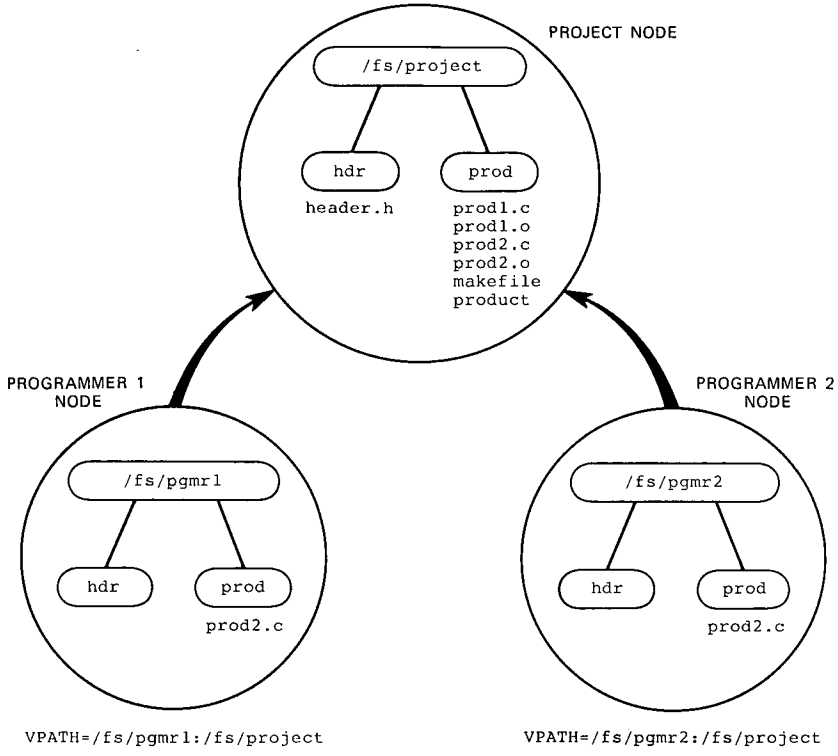

Fig. 4—Independent software views.

arrows. `/fs/project` contains all the files (source, objects, and intermediate objects) necessary to generate `product`. The other viewpath, `/fs/pgmr1:/fs/project`, is the viewpath for a developer named programmer 1, who wishes to generate a software view that includes personal changes in the node `/fs/pgmr1` in addition to the released software in the shared node `/fs/project`. The only connection between the nodes is the logical one defined by the viewpath specification. Similarly, programmer 2 keeps personal changes in the node `/fs/pgmr2` and includes the project node in a different viewpath specification. This illustrates how viewpaths express individual software views for two developers. Both programmer 1 and programmer 2 may have private versions of files in their respective nodes and make changes independently of each other. Each developer will have a copy of the files that they wish to change, and the file(s) will have the same relative file name as in the `/fs/project` node but with a different full path name. The developer's version of the file will, in effect, overlay the instance of that file in the project node. Note that one developer's changes are invisible to all other developers since the changes appear only in that developer's private node.

III. DESCRIPTION OF BUILD

The `build` tool makes available the capabilities of `make` within the context of a software view. To use `build`, the desired viewpath must be declared, either on the command line or using the `VPATH` environment variable. `Build` must be invoked from a directory within the first node in the viewpath. The `build` tool uses the viewpath to resolve all relative file references, both to the description file itself and to the target and dependency references within the description file. If `build` does not find a file in the first node, it looks in the same directory in successive nodes in the viewpath until the file is found or the last node in the viewpath is reached. If the viewpath contains only a single node, `build` resolves all file references relative only to the current directory. Hence, a `build` with a single node viewpath is equivalent to `make`.

The `make` tool rebuilds a target file if the file does not exist or if it is dependent on a file that has a later modification date. The `build` tool rebuilds a target if either of the above criteria holds, or if it is dependent on a file existing in an earlier node in the viewpath. This additional criterion is necessary for `build` to ensure that a correct version of the product is produced when a target file is put in a node with a modification date later than the file that is in an earlier node in the viewpath, upon which it depends.

In preparation for rebuilding targets, files that the target depends on and that are not in the first node are temporarily added to the first

node. `Build` does this by using the *UNIX* link command, `ln`, if the two nodes involved are in the same *UNIX* file system; or the copy command, `cp`, if they are in different file systems. The generation of the target files depends on these files being made available in the first node. Since these files are determined from the dependency lists in the makefile, the completeness and correctness of makefiles is necessary for `build` to affect changes to the software. If there are errors of omission in makefile dependency lists, they reveal themselves during the building of a target as files that are missing—i.e., not added to the first node. After the targets are built, all files copied or linked into the first node are removed by `build`.

In addition to enabling a number of developers to share a set of files to achieve individual software views, `build` can also be useful to the individual developing a product. A developer often wants to make and test changes to a program without destroying a previous version of the program. The `build` tool allows the developer to conveniently produce a new version without having to save the old one or rewrite the makefile. The developer simply creates a new node, places it in any modified files, sets the viewpath to look first in the new node followed by the original node, and then invokes `build`.

IV. EXAMPLE

We now return to the nodes shown in Fig. 4, and consider how `build`, using the makefile shown in Fig. 3, handles the particular files in the nodes. `/fs/project` contains all the files (source, objects, and intermediate objects) necessary to generate `product`. This means that at one time `build` was used to generate `product` in this node. Since there would have been only one node in the viewpath, using `build` in this situation was identical to using `make`. Assume that the viewpath for programmer 1 is `/fs/pgmr1:/fs/project` as shown in Fig. 4.

If programmer 1 changes the file `prod2.c` and runs `build` from within the directory `/fs/pgmr1/prod`, `build` looks for a description file named `makefile` and finds `/fs/project/prod/makefile`, since there is none in the node `/fs/pgmr1/prod`. As `build` processes the makefile, it determines first that `product` depends upon `prod1.o`, which in turn depends upon `prod1.c`. `Build` locates both:

```
/fs/project/prod/prod1.o
/fs/project/prod/prod1.c
```

In `/fs/project/prod`, `prod1.o` is newer than `prod1.c`. Consequently, no regeneration of `prod1.o` occurs. `Build` then deter-

mines that `product` depends upon `prod2.o`, which in turn depends upon `prod2.c` and `../hdr/header.h`. The `build` tool locates:

```
/fs/project/hdr/header.h

/fs/project/prod/prod2.o

/fs/pgmr1/prod/prod2.c
```

Since `/fs/pgmr1/prod/prod2.c` is newer (or is in an earlier node) than `/fs/project/prod/prod2.o`, `prod2.o` needs regeneration. The `build` tool links or copies any files necessary for regeneration into the lowest node in the viewpath. In this case, `/fs/project/hdr/header.h` is linked to `/fs/pgmr1/hdr/header.h`. Now,

```
cc -c -O prod2.c
```

is invoked by `build`, producing `prod2.o` in the developer's private node. After `prod2.o` is generated, `/fs/pgmr1/hdr/header.h` is unlinked. Because `product` depends upon `prod2.o`, which was just generated, `product` is regenerated. Again, any dependents of `product` must be in the developer's node. `/fs/project/prod/prod1.o` must therefore be linked to `/fs/pgmr1/prod/prod1.o` before the following command can be executed:

```
cc prod1.o prod2.o -o product
```

This produces the executable object `product` that is located in `/fs/pgmr1/prod/product`. `Build` then unlinks `/fs/pgmr1/prod/prod1.o`.

A similar scenario occurs for programmer 2, since that developer also has a private version of `prod2.c`. Each developer is thus able to generate a private version of `product` containing only the changes they wish to test. The only files that each developer retains in their private node are `prod2.c`, `prod2.o`, and `product`, as shown in Fig. 5. Other files that they have not changed remain in the project node as shared files.

V. SAMPLE PROJECT USE OF BUILD

In the *UNIX* operating system environment, a *system of software* consists of a collection of source, object, and executable object files placed in some orderly fashion within a collection of directories (*directory structure*). A set of cooperating makefiles are placed throughout the directory structure, each specifying the instructions and the files that are needed to construct a given target file. `Build` is used to construct the targets for the first time from the original source files

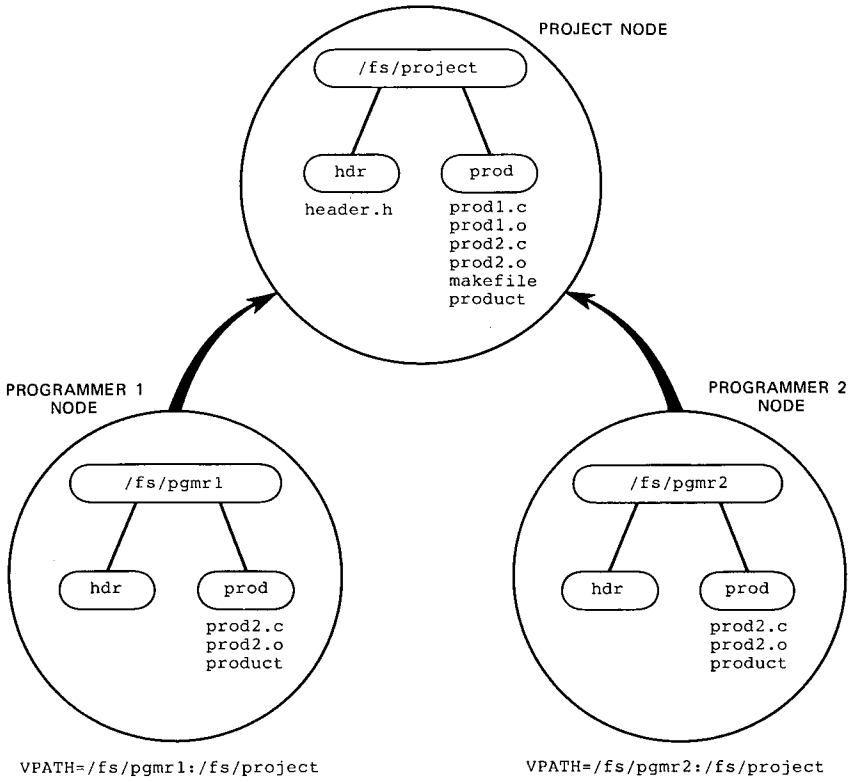

Fig. 5—Contents of nodes after executing `build`.

or to propagate changes made to a subset of the source files within the context of a complete constructed system.

There are a number of projects at AT&T Bell Laboratories that are currently using `build` as the primary construction tool. The AT&T *UNIX* Real-Time Reliability (RTR) project has over 100 developers dealing with more than 8,000 source files, for each of three major versions of the system.[2] The `build` tool is used throughout all phases of the software generation.

1. Unit testing. On each machine assigned to developers on the project, a fully populated, official node of each major version of the system is provided as a stable base for unit testing individual developer changes to the system. Using `VPATH`, developers define their viewpath to be one or more private nodes followed by the official node.

2. Integration testing. At the project level, changes constructed with the official node in the viewpath and generated by many developers are combined by accumulating them in a single test node in order to provide versions of changed products for official testing. In the case

where several developers have made changes to a single source file, the source administration system used in the project provides a source file version containing the combined changes.

3. System release generation. Changes to be combined for an incremental release of one of the system versions are accumulated in a single node (as in integration testing). These are built and system tested with the official node for the previous version of the system in the viewpath before being released.

The `build` tool is the only construction tool used in the *UNIX* RTR project, and has been extremely successful at minimizing multiple copies of files, something that can easily become a serious problem in a project of that size. Independent unit testing by developers is easily accomplished in this environment.

VI. BENEFITS

`Build` reduces the number of multiple copies of files through the use of a globally shared node containing a full set of the software, and supports independent unit testing of changes by individual developers. An additional and important benefit is that when multiple nodes participate in building software, the correctness of dependency lists in the makefiles is enforced, since `build` is otherwise unable to make all necessary files available in the first node of the viewpath for target generation. The completeness of makefiles is similarly tested, since the components necessary to construct a particular product must be made available by `build` to the developer's private node. Confirming the correctness and completeness of the makefile is absolutely necessary for a reliable software construction process.

VII. SUMMARY

`Build` is an extremely useful extension to the *UNIX* operating system `make` tool for constructing and changing software. The `build` tool automates the steps between editing and testing within the framework of individual software views. Individual software views allow a number of developers to easily make and test changes to a set of shared files without interfering with each other. `Build` also provides a check on the completeness and correctness of makefiles used to specify software construction steps. Many projects at AT&T Bell Laboratories are currently using `build` with much success.

REFERENCES

1. S. I. Feldman, "Make—A Program for Maintaining Computer Programs," Software—Practice and Experience, *9*, No. 4 (April 1979), pp. 256–65.
2. B. R. Rowland and R. J. Welsch, "Software Development System," B.S.T.J., *62*, No. 1, Part 2 (January 1983), pp. 275–89.

AUTHORS

Verlyn B. Erickson, B.A. (Mathematics/Physics), 1968, Augustana College; M.S. (Computer Science), 1970, University of Wisconsin, Madison; Mathematics Research Center, University of Wisconsin, 1969–1972; Engineering Computing Laboratory, University of Wisconsin, 1970–1977; AT&T Bell Laboratories, 1977—. Mr. Erickson has worked in various software support areas, including 1*ESS*™ Laboratory Support, AT&T 3B20 DMERT Laboratory Support, and, most recently, the Software Development environment for the 3B Processor line.

John F. Pellegrin, B.A. (Mathematics), 1965, Occidental College; M.A. (Mathematics), 1967, Arizona State University; AC Electronics Defense Research Laboratories, Goleta, CA, 1967–69; Ph.D. (Mathematics), 1972, Arizona State University; AT&T Bell Laboratories, 1972—. Mr. Pellegrin has worked in various development support areas, including hardware logic simulation, *ESS*™ software development support, software development methodology, and, currently, Software Development Systems for the AT&T 3B Processor line.