# ‘S’ IS FOR ‘SOURCE’: The Role of the Build System in Configuration Management

**Anders JOHNSON,** NVIDIA Corporation, ajohnson@nvidia.com
**Gary HOLT,** Safe-View Incorporated, holt-makepp@gholt.net

**Abstract:** *Perforce bills itself as a “Software Configuration Management System.” This is a bit of a misnomer, both because it is also well-suited to hardware development, and because there are several needs commonly considered part of the configuration management problem that it does not address. We explore the role of the build system (specifically, a build system based on Makepp) in addressing some of these needs in the context of a large integrated circuit development project using Perforce for* Source *Configuration Management (SCM).*

## 1 Configuration Management Defined

Although there is no single well-accepted definition of *Configuration Management*, our experience has been that most people associate it with the following needs[1]:

- Source Control
- Defect Tracking
- Compilation
- Dependency Tracking
- Testing
- Packaging
- Release Engineering
- Variant Management
- Customer Installation

Perforce concerns itself primarily with source control.[2] Perforce does support elements of defect tracking,[3] but in practice few installations actually make use of this capability, opting instead for some unrelated tool or for informal communication. Perforce inter-file branching can be used for variant management, but this approach has been shown to scale poorly, especially in the number of variants and in the distance from the common ancestor.[4]

Furthermore, Perforce is found to be well-suited to integrated circuit (IC) design, in particular because it scales well in the amount of source data.[5] If you are a software person trying to understand the semi-custom IC design landscape, imagine developing any large program as you normally would, except that it costs $1,000,000 and takes 3 months to burn the first copy of each new CD, and if you don’t run the program from a CD then it’s 1,000,000 times slower and might not have all of the bugs. While the behavioral complexity of an IC is no match for that of a large program, the additional attention that must be paid to implementation, optimization and verification more than compensates in terms of source complexity. Hardware and software development activities may seem superficially unrelated, but the underlying configuration management needs are fundamentally similar.

From this, we submit that the role of Perforce should not be considered limited to *Software*, but for the most part it should be considered limited to *Source*. Fortunately, this shift in thinking requires no change in the *SCM* acronym that describes Perforce’s *Source* Configuration Management role.

---

[1] http://www.cs.utah.edu/dept/old/texinfo/cvs/FAQ.txt 1A.5

[2] This is not a criticism of Perforce, because separation of responsibilities is generally a Good Thing. In particular, we believe that the focus of Perforce on source control contributes to its serving that need exceptionally well.

[3] There is good reason to couple defect tracking and source control, because tracking fixes across branches is nontrivial. See Wingerd, Laura, “A Bug’s Life,” 2003 Perforce User Conference.

[4] Jackson, Peter and Richard Brooksby, “Changing how you Change,” 2003 Perforce User Conference; Johnson, Anders, “Codeline Management for Evolutionary Development,” 2003 Perforce User Conference, §11.

[5] Sikand, Shiv, “Integrated Software Configuration Management for Hardware Design,” 1999 Perforce User Conference.

## 2 The Build System Defined

We define the *build system* as any part of the deterministic, automated process on the development platform of producing files that are not always obtained from the source control system, or of ascertaining properties of source and/or generated data.[6] As a rule, each configuration management role should be automated if possible,[7] and we generally find the following needs amenable to automation:

- Compilation
- Dependency Tracking
- Testing
- Packaging
- Release Engineering
- Variant Management

Therefore, these needs should be addressed by the build system. Not surprisingly, build tools such as Ant, Jam/MR and the various flavors of Make lend themselves to creating an integrated system that automates (or at least facilitates) these needs. In this paper, we will focus on a lesser-known build tool called Makepp,[8] which is used at NVIDIA, among other organizations, and which is in many ways superior to alternative build tools.

Figure 1 illustrates our view of the configuration management universe. Note that the separation between the roles of the build system and those of Perforce is along the same axis as the distinction among configuration management needs, and is orthogonal to the distinction between the hardware and software domains.

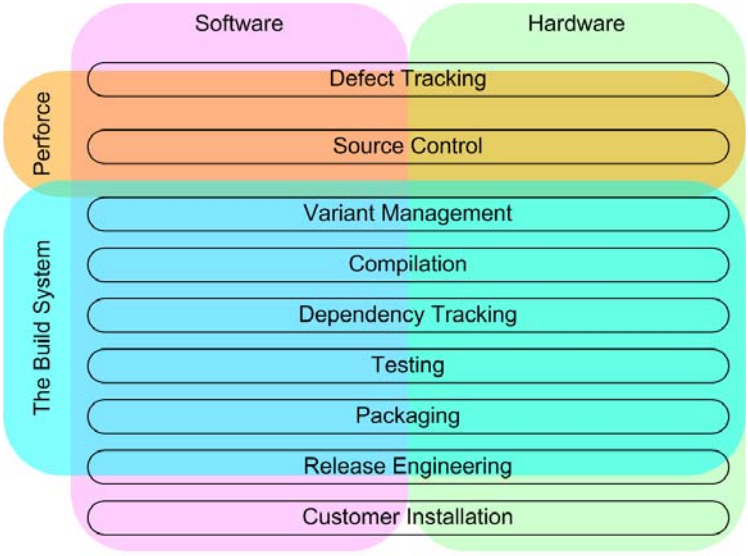


Figure 1. The Configuration Management Universe

Specifically excluded from our definition of the build system are the processes for compiling and installing on target platforms, because although they represent important problems, their inclusion would muddle the build system concept. For example, files included in a "source distribution" may actually be files generated on the development platform that are expensive or impossible to build on the target system (in particular, because they are built with tools that are not available on the target system). Furthermore, source files may be packaged with files for compiling them with or for facilitating their integration into one or more build tools other than the ones used on the development platform.[9] However, the process of *testing* that compilation and installation on the target system works properly can (and probably should) be part of the build system proper.

The only remaining configuration management need that we can think of is that of defect tracking. While neither of the authors have much experience with **p4 jobs**, it seems reasonable to expect that the defect tracking need would best be addressed by a tool that integrates with the Perforce defect tracking facilities, especially if branching is or will be employed.

[6] The term *process management* is sometimes used to describe this role, but that term is slightly more general in the sense that it may extend to include manual processes.

[7] Bowles, Jeff A., "Handoffs 101," 2003 Perforce User Conference.

[8] http://makepp.sourceforge.net/

[9] In particular, open source tools traditionally ship with some build facility that will not only portably clean build but also build incrementally, provided that the dependencies, the environment, executables in the path, command-line variable settings, etc. don't change.

## 3 The Incremental Build

Every build tool worth its salt support incremental building, wherein steps of the build process are avoided if the build tool can trivially determine that they should not substantively affect the state of the workspace.[10] However, most build systems (especially for hardware development) do not have a high-reliability incremental build.[11]

In addition to saving time by building incrementally, a good build tool offers other advantages over a script that executes every step each time whether it is necessary or not. Build tools define a partial ordering of commands, *i.e.*, the requirement that some commands be completed before others begin. This makes it possible to determine whether a given reordering of the commands is safe. More importantly, it also permits dynamic parallelization of the build steps to make optimal use of currently available resources.

On the other hand, if the partial ordering is incorrect, allowing the build tool to choose the sequence of steps can lead to nondeterministic results. A build tool that works *most* of the time can be worse than no tool at all because it can be extremely difficult and time-consuming to diagnose and correct incorrect builds.

Living with an unreliable incremental build is sometimes acceptable if the clean build time is short (less than a minute), because making the incremental build reliable may not be worth the investment of time, especially if the chosen build tool is not well-suited for the task. However, this solution scales poorly, because both the build time and the probability of an inaccurate build increase with the total code size of whatever level of integration you're building. For compiled software in particular, the clean build time tends to become quadratic in the amount of source code, because every object file will tend to depend on most of the header files unless conscientious design practices are enforced to prevent that from happening.[12]

One way to overcome an unreliable build is for users to employ expert knowledge of the system to manually remove specific generated files only as necessary in order to build correctly. Unfortunately, this represents a tremendous burden on the users, and ironically it is precisely what build tools are supposed to do.[13] Furthermore, manually removing only a few generated files is usually not feasible after **p4 sync**, because it requires much more diligence and attention on the user's part to understand the affect of other people's unrelated changes than it does to understand the effect of his own local changes. For this reason, a clean build is usually required after **p4 sync**.

An additional risk of using heuristics to overcome an unreliable incremental build is that the build may *succeed* only to produce *incorrect* generated files (whose contents differ substantively from what they would be at the completion of a clean build). Such occurrences tend to be extremely difficult and time consuming to diagnose, and the fact that they can be avoided by clean building makes them all the more frustrating.

For these reasons, we believe that a reliable incremental build is of inestimable value to the development of a large project. Unfortunately, as we will see, using most popular build tools, the difficultly of achieving a reliable incremental build increases with the size of the project. At the same time we take solace in the knowledge that achieving a reliable incremental build using Makepp is relatively easy, essentially independent of project size.

## 4 Things that Make the Incremental Build Unreliable

There are a large number of specific issues that can cause the incremental build to be unreliable. GNU Make tends to suffer from all of them, but each other build tool may address some of them with varying degrees of success.[14]

Perhaps the most common cause of an unreliable incremental build is missing dependencies. There are many ways that dependencies can be omitted. For example, if recursive Make is employed, dependencies between files in different directories are neglected. Implicit dependencies (dependencies that come into existence due to the contents of another file, for example, dependencies of object files on header files) may be ignored, requiring the maintainer of the build control files to track them in redundancy with the source files.[15]

Dependencies on the build control files themselves (for example, the command used to generate each target) may be ignored. Dependencies on the platform and/or the environment may be ignored. Dependencies on the executable(s) used to generate a target, as well as the libraries that they load, may be ignored.

---

[10] In particular, changes in generation timestamps are deemed not substantive.

[11] Miller, Peter, "Recursive Make Considered Harmful," AUUGN '97.

[12] Lakos, John, "Large-Scale C++ Software Design," Chapter 6.

[13] It is an unfortunate reality that those responsible for making build-related decisions are most often those who are most familiar with, and therefore least annoyed by, legacy build systems whose incremental build is unreliable.

[14] Several issues with GNU Make are identified in Miller, *op. cit.*

[15] Both Ant and Jam/MR address the implicit dependency and cross-directory dependency issues.

Modifications to the build control files for the purpose of overcoming missing dependencies tend to be themselves unreliable and inefficient, as well as difficult to maintain. For example, in GNU Make implicit dependencies are often tracked with `.d` files,[16] but Anders has observed complex cases in which it does not work after a header file is deleted, even when there is a null rule to build the header.[17] As another example, adding a dependency of every target on the makefile causes an unnecessary clean build every time the makefile is touched, but it does not cause affected targets to be rebuilt when rules change due to command-line variable overrides. As a final example, making the makefile flat in order to avoid missing cross-directory dependencies requires the entire build system be loaded, even to build only trivial targets; and it also requires writing the makefile and its include files in a peculiar manner for which Make was not designed.

Another cause of unreliable incremental builds is that stale targets (files that were generated by the build system in the past, but the build system has since been modified such that they are no longer generatable) may appear to the build system as files obtained from Perforce. This can manifest itself in a number of ways, perhaps the most common of which is that linking together all the files that match `*.o` will incorporate stale object files, and the results can be very strange and unexpected if they contain symbols that are defined in other object files as well (which in particular happens when some of the symbols in the stale object file are migrated to a different object file).

Sometimes problems occur when a target to be built already exists and the program that builds it doesn't handle it properly. For example, its permissions may not allow it to be overwritten. Worse, it could be a soft or hard link to another file somewhere else, and overwriting it can cause "spooky actions at a distance" which can be difficult to isolate. Well-behaved programs will remove their targets before generating them, but rules that utilize programs that don't do this should be prepended with `$(RM) $@` (in Make lingo) to avoid this problem. Similarly, many build tools (other than Makepp) will treat a target left behind by a failed rule as an up-to-date file, and thus one may have to suffix invocations of ill-mannered programs with `|| (s=$$?; $(RM) $@; exit $$s)` to avoid the issue.

## 5 Miscellaneous GNU Make Issues

There are also a number of notable issues not specifically related to the incremental build that cause GNU Make to be a suboptimal build tool.[18]

Wildcards in GNU makefiles don't match targets that aren't already built. This causes the *clean* build to fail, requiring the build tool to be re-invoked, possibly several times, before it succeeds.[19]

GNU Make recompiles a dependency when its timestamp changes, even if its contents didn't change.

GNU Make doesn't recognize different paths that refer to the same file as such.

GNU Make doesn't accumulate the output of individual rules when it runs in parallel, and the interleaved output is typically useless.

## 6 Enter Makepp

The NVIDIA Verilog register transfer level (RTL) build system underwent a thorough re-engineering effort in 2003. Options that we seriously considered were:

- Reorganizing the existing recursive GNU Make system.
- Migrating to Ant.
- Migrating to Jam/MR.
- Migrating to Makepp.

As NVIDIA is a dyed-in-the-wool Perforce outfit, Clearmake[20] was not an option. E-Make[21] warranted some attention, but it was deemed too immature for us to adopt it as a closed system.

---

[16] GNU Make texinfo: "Generating Prerequisites Automatically."

[17] It seems always to work in simple cases, though, and the problem has not been isolated.

[18] As noted before, other build tools address some of them with varying degrees of success.

[19] David Whipp ironically calls this a *truly* incremental build, because each attempt gets incrementally closer to success.

[20] Clearmake is the build tool that is integrated with ClearCase.

[21] Electric Cloud

GNU Make was easy to eliminate in principle, because it was clearly the worst alternative in terms of actual behavior. However, its establishment as the legacy system made it difficult to displace, and it lingers as part of the build even today.

Ant and Jam/MR solve a great number of the problems with GNU Make, but they were both deemed to difficult to extend beyond their software roots. In particular, neither of them currently detects all of the implicit dependencies even in Verilog, and NVIDIA uses a number of homegrown tools and relatively obscure third-party tools, many of which have idiosyncratic rules for determining implicit dependencies.[22] Furthermore, the syntaxes of the build control files were simply too foreign to any of the developers, almost all of whom were familiar with (if not fluent in) Make.

Makepp, on the other hand, seemed to go the farthest in solving the myriad GNU Make deficiencies, and was relatively easy to extend, both from within makefiles and, being written in Perl, in terms of modifying the Makepp source code. It also solved, or had the potential to solve, a number of efficiency and maintainability issues that we hadn't even identified. While the existing user base of Makepp was by far the smallest, it was a clear winner for our application.

## 7 The Makepp Solution

Makepp uses a syntax which is almost identical to GNU Make (most simple makefiles will work without modification), but circumvents all of the above problems because it uses different data structures and determines whether a rebuild is necessary using different conditions.

First, it avoids recursive Make and the problems associated with it by loading all the necessary makefiles into memory simultaneously. It parses the makefiles and determines what targets can be built with each makefile. It records the actions required to build each target in a data structure that mirrors the directory hierarchy on disk. This data structure allows Makepp to represent cross-makefile dependencies. For example, suppose `subdir1/Makefile` has a rule to build `xyz`, which depends on `../subdir2/abc`. `subdir2/Makefile` has a rule to build `abc`, which depends on `../subdir1/pdq`. Because both makefiles are loaded simultaneously, Makepp can first execute the rule from `subdir1` that builds `pdq`, then cd to `subdir2` to execute the rule to build `abc`, then cd back to `subdir1` to build `xyz`. (See Figure 2, right.)

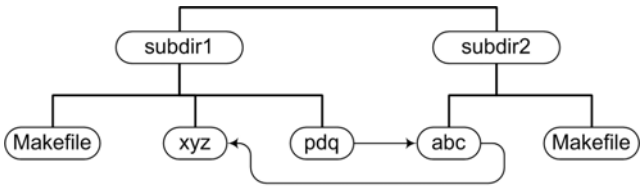

Figure 2. Cross-directory Dependencies

A complicated directory hierarchy, particularly one with mutual dependencies as in this example, can be extremely difficult to handle correctly using recursive invocations of Make in the standard Unix Make paradigm. With Makepp, it's handled for you automatically.[23]

Makepp builds up this structure by populating it first with the files that already exist. Then it determines what files can be built from those using the rules in the makefiles, applying whatever pattern or wildcard rules are present. It keeps iterating this process until no additional files can be built. So by the time Makepp has finished loading all the necessary makefiles, it has a complete list of all the files that can be built in the directories containing those makefiles (even if those files aren't actually requested). This turns out to be useful, because in Makepp `*.o` matches not only all the `.o` files which currently exist, but also those files which do not yet exist but can be built. Use of wildcards in makefiles sometimes greatly shortens them and often avoids the need to list explicitly every file.

Once Makepp has loaded all the makefiles, it proceeds to build the requested targets. Makepp has two critical improvements over GNU Make in this regard: automatic scanning for include files, and a much more reliable algorithm for determining when to rebuild.

A key problem with build systems is to determine the implicit dependencies such as `.h` files. Instead of requiring a special syntax in the makefile (*e.g.*, including `.d` files generated by Gcc), Makepp automatically knows how to scan for include files. When requested to build a file, it first attempts to parse the shell command. If it is a compilation command that it understands, Makepp finds the include path (*e.g.*, all the **-I** options if it's a C compiler,

[22] A facility for predicting the implicit dependencies of a command is called a *scanner*.

[23] Of course, you want to avoid such complicated dependencies where possible, so that you coworkers don't shoot you when they try to understand what you have done. However, sometimes cross-directory dependencies are impossible to avoid.

and the **-L** options if it's a link step). It also finds any additional files that aren't in the explicit dependency list. (For example, it attempts to find all the libraries specified via **-lxyz**. Also if the command is not a compilation command, but a script that resides in your build directory, it marks the script as a dependency so if you change your script the command is re-executed.) It then scans through the source code looking for `#include` statements (or the equivalent in whatever language). It determines the location of each include file (optionally removing any stale generated files that would be used), builds the file if necessary, and then scans those include files. The scanned info is cached so that if the files do not change, Makepp does not need to rescan them. Currently support for C, C++, and Verilog compilers is distributed with Makepp; the mechanism is extensible using Perl.

In our experience, this scanning mechanism is more reliable than using the **-M*** options of Gcc and similar options of other compilers. One reason for this is that Makepp will correctly build `.h` files that don't yet exist, even if they will not be in the current directory but in some other directory in the include path. By contrast, the **-M** options assume that non-existent `.h` files are in the current directory.

Once Makepp has found and built all the files that a given target depends on, Makepp by default uses the following criteria to determine whether to rebuild a file[24]:

1. If any dependency's date or size has changed since the last time this build command was executed, then Makepp rebuilds. Makepp looks for any change, so by default it will rebuild if you replace a file with an older version. This also allows it to avoid problems where the time is not correctly synchronized across file servers.
2. For source files, and other files you request, Makepp will also compute an MD5 checksum of the file. If the dependency's date has changed but the checksum has not, Makepp will not rebuild the target. Makepp's checksums ignore comments and syntactically insignificant spaces/tabs in source files, so you can add comments or re-indent without triggering a rebuild.
3. If the output file has changed since the last time the build command was executed (*e.g.*, someone manually executed a command that changed it), then Makepp rebuilds. This guarantees that the output is what the makefile specified and won't be corrupted by commands not executed under control of the build system.
4. If the command has changed (*e.g.*, if you added an extra **-D***symbol=value* compiler option), Makepp rebuilds. This eliminates a large category of subtle build defects, where some modules are built with the wrong compilation options.
5. If the architecture has changed (*e.g.*, last time you built on a Sun and this time you're building on Linux), Makepp rebuilds.

This build comparison algorithm requires storing information from the last build. Makepp stores this in a separate text file which it rereads later. If the file was `/path/to/xyz`, then Makepp stores the build information in `/path/to/.makepp/xyz.mk`. Other programs can also read this text file, and we have also found it convenient to view these files by hand occasionally because they tell exactly what command options were last used to compile each file.

## 8 Keeping the Source and Build Systems Separate

While there are legitimate reasons to couple the source control system and the build system, we believe that it is architecturally preferable to keep the source control and build systems completely separate, and achieve in other ways the benefits that could be obtained by coupling them. For example, instead of storing generated files in the source control system, cache them using the build system. (See §12, "Build Caching with Makepp.") As another example, instead of having the build system rely on the source control system to inform it when a build command reads a file or when a source file is updated,[25] customize the filesystem to do so. [26]

[24] Many of these build comparison ideas were taken from Bob Sidebotham's cons utility; see http://www.dsmit.com/cons.

[25] ClearCase, Vesta

[26] E-Make from Electric Cloud does this to detect dependencies, but Makepp does not.

Given that your source control system is Perforce, this basically boils down to not having anything in your build system that does any **p4** commands, even read-only commands. The main reasons for this are:

- To avoid vendor lock-in.
- To ensure that the build works when you are off-line from the Perforce server.
- To make it easier for your customers and partners who don't use Perforce to reproduce the build.

Avoiding **p4** commands presents a bit of a challenge for release engineering, but we have found that the SCM information that needs to be incorporated into a release can generally be obtained from ID keywords in some dummy file that is required to be checked-in with the final changelist prior to a release. The automation of procedures that absolutely must rely on Perforce (for example, verifying that the dummy file was checked in) should be decoupled from the remainder of the build system as much as possible; specifically, such procedures should not be invoked from within a build tool.

## 9 Client Options

Choosing the right client options is important for ensuring that the build system interacts properly with Perforce. In some cases, the default options, which are designed to maximize safety, will interfere with the build system.

- **nomodtime** (the default)

In order for the build system to recognize that a file has changed when you sync to a version that is older than the targets that depend on the file, you may need to set **nomodtime**. This is usually not necessary for Makepp, because it remembers the exact timestamps of all dependencies when it builds a target, but it is especially important for GNU Make.

- **clobber** (*not* the default)

In order to handle the case wherein a generated file becomes stale, and a source file is added in its place, **clobber** needs to be set. Otherwise, manual intervention may be required when you **p4 sync**.[27]

- **rmdir** (*not* the default)

In order to handle the case wherein a directory is deleted and a generated file is added in its place, **rmdir** needs to be set.[28] Note that if the directory contains stale generated files when the last source is removed, then Perforce will not remove the directory, so you still might be out of luck. You can't avoid this by removing stale generating files before **p4 sync**, because the files might not become stale until after **p4 sync**.

## 10 Variant Management via the Build System

In 2003, Anders, and Peter Jackson along with Richard Brooksby independently reported that branching is not a suitable general technique for variant management.[29] Anders' experience at NVIDIA confirms this finding. In fact, attempts at using branching for variant management have given branching such a bad reputation in general that it has become essentially impossible to justify branching for legitimate roles, such as development.[30]

We have found that variant management using the build system is a far preferable approach, and moreover it is an approach with decades of prior art. This does not mean eschewing branching completely, but it does imply that even in a release branch, all the variants should build (although only the variant to be released will necessarily be required by the branch policy to be of the highest possible quality).

[27] This is often a symptom of editing a generated file, which is bad practice, but it does happen, sometimes legitimately.

[28] This is a regular occurrence at NVIDIA.

[29] Johnson, *op. cit.*, §11; Jackson, *op. cit.*

[30] Vance, Stephen, Advanced SCM Branching Strategies, 1998 Perforce User Conference.

As a general principle, variability is best modeled in source code at the highest level of abstraction available, such that the amount of redundant source data is minimized.[31] In other words, it is advisable to go to great lengths to ensure that the commonality among the variants is represented in only a single place in the source code. In practice, this means that every nontrivial source file should be invariant.[32]

On the other hand, while invariant generated files are somewhat preferable to variant generated files, all else being equal, you ultimately can't have variation if all of the files are invariant. Furthermore, practical considerations such as size and efficiency may force variation to be bound early in the build flow, such as at compile time. In particular, if the format of an input file does not allow for sufficient abstraction to effectively express variability without excessive redundancy, then one can make that input file a variant generated file based on some other source file processed by a program (such as Cpp, M4, Awk, Sed or Perl). Note that this approach may not work if the build tool discriminates between source and generated files, for example, if the build system accesses the source control system, or if it uses wildcards in GNU Make.

As a rule, each variant of each generated file should build in a different location in the workspace. This has a number of advantages:

- It avoids discarding generated files for a first variant simply because a second variant is built. If this happens, then when the first variant is built again the files need to be rebuilt from scratch.
- It allows multiple variants to be visible simultaneously, which is necessary in particular to test interoperability among the variants.
- It allows the designation of the variant to be localized. In particular, a variant of one file may depend on a different variant of another file.
- It permits a specific variant to be requested of the build tool.

As a shorthand, it is advisable to have shared phony targets that map to (*i.e.* depend on) one or more variant targets according to a configuration file in the workspace or to the value of an environment variable. This allows users to request targets without having to type the variant name for each of them.

Variability tends to increase over the life of a project. That is, files that were once invariant gradually accumulate additional responsibilities, and each of these accumulated responsibilities may entail variability. Because of this phenomenon, it is a good idea to make variant-ness late binding. One way to accomplish that is to assume that *every* generated file is a variant file, even if it currently isn't. Implementing this is simply a matter of ensuring that there is a variant location to build every file, and of making all the dependencies on the generated file refer to that variant location. Unfortunately, this discipline introduces an asymmetry between source and generated file, but that asymmetry can be remedied by using Makepp repositories.

## 11 Using Makepp Repositories for Variant Management

Makepp supports *source repositories*, wherein a file from a source directory is projected into a destination directory via a symbolic link if the file does not already exist in the destination directory and there is no rule to build it. The concept is similar to `VPATH` in GNU Make, except that `VPATH` is implemented by changing the command line invocation of programs that read the source file, whereas repositories are essentially transparent to the program because they are implemented with symbolic links.

Source repositories can be used to regularize dependencies by allowing all of them to refer to variant locations, whether or not they are source dependencies or generated dependencies.[33] This is simply a matter of making the invariant location the source directory of the repository, and the variant location the destination directory.

At NVIDIA, each potentially reusable (*i.e.* independently testable) component is called a *unit*, and is assigned a directory. Source files are located inside the unit directory. Every generated file is located inside one of many variant subdirectories that are located just below the unit directory. This scheme was chosen partly because it aligned well with the legacy build system, and it has disadvantages as well as advantages.[34] In particular, it

[31] This principle is known as "The DRY Rule." See Hunt, Andrew and David Thomas, "The Pragmatic Programmer," Chapter 2.

[32] In this context, *invariant* means remaining the same from variant to variant, but not from revision to revision.

[33] A side-benefit of depending on repositories for all source files is that missing dependencies are more likely to be detected by virtue of a missing symbolic link.

[34] An eminent alternative would be to locate source files in a `src` directory below the unit directory.

introduces an ambiguity between variant subdirectories and subdirectories containing source data – the latter need to be projected into variant directories, whereas the former do not. (See Figure 3, right.) A central file disambiguates the situation by listing the unit directories, as well as the possible names of variant subdirectories.

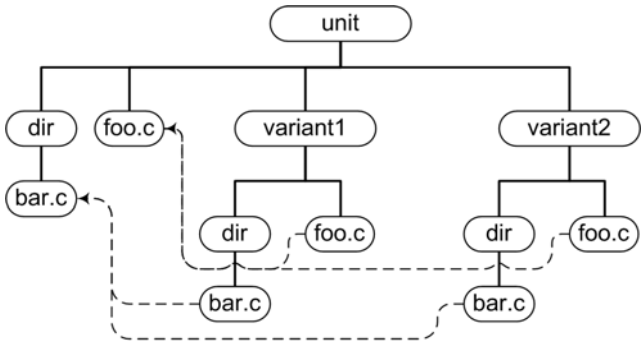


Figure 3. Source and Variant Directories

Because Makepp has difficulty dealing with generated directories, all of the variant directories are created at the beginning of each build before Makepp has a chance to reference them. This is implemented by wrapping Makepp in a script that adds an option to load the top-level makefile before doing practically anything else. Although the number of generated directories is large (the NVIDIA build has between 100 and 1000 variant directories) and directory creation over NFS is particularly slow, parallelizing this step allows it to verify the existence of over 10,000 directories per second in the NVIDIA compute environment. Actually creating the directories takes about 4 milliseconds per directory whether or not the process is parallelized, but this is still tolerable, and it matters only on the very first build of a workspace.

The same procedure that creates the directories also stores the variant name of each variant directory in Makepp’s filesystem database (in Perl memory). It then arranges to load the unit directory as a repository for each of its variant directories when the variant directory is first referenced by Makepp, by assigning to `$FileInfo::directory_first_reference_hook`. Because each makefile is itself a source file that gets projected into its variant directories, it uses in-line Perl code to determine the variant to which it currently applies, by looking up the current directory in the database. Normally, if the makefile applies to the source directory, then it will define only rules for phony targets. Otherwise, it will define rules wherein all the dependencies are located in variant directories, usually of the same variant. The name of the current variant may also serve as a root cause of variation, for example, by introducing it as a **-D** option to a compiler command line.

```
TOP := $(dir $(find_upwards TOT))
include $(TOP)/make/variant.mk
ifdef VARIANT
 TOP_MODULES := my_module
 MODULE_PATH := . \
  $(TOP)/other_unit/$(VARIANT) \
  $(TOP)/shared_library/$(VARIANT)

 simv: $(TOP_MODULES:%=%.v)
        vcs –o $@ $(MODULE_PATH:%=-y %) \
         $(MODULE_PATH:%=+incdir+%) \
         +define+$(VARIANT) \
         +vcs+lic+wait -Mupdate $(inputs)
else
 $(phony simv): $(VARIANTS:%=%/simv)
endif
```

Figure 4: Simplified Example Unit Makeppfile

Figure 4 shows a simplified example of a unit Makeppfile. (In reality, there are many targets, and common functionality is aggressively centralized in include files.) The first step is to determine the relative path to the top of the workspace, which is uniquely identified by virtue of containing a file named `TOT` (for “top of tree”). Then, we include the `variant.mk` file, which defines the variables `$(VARIANT)` and `$(VARIANTS)`. (In Makepp, variables may be assigned from arbitrary in-line Perl code.) The `$(VARIANT)` variable will contain the empty string if the Makeppfile is being loaded for the source directory. (Like GNU Make, Makepp treats the empty string as undefined for the purposes of `ifdef`.)

If the variant is defined, a rule to build the `simv` target is defined. Because Makepp has a built-in scanner for VCS,[35] dependencies on included files and files defining instanced modules are automatically detected. The **+vcs+lic+wait** option prevents failures due to the exhaustion of licenses, at the expense of waiting for one to become available if necessary. The **-Mupdate** option permits VCS to utilize previously computed information about the input files.[36]

If the variant is not defined, then the list of variants selected by the workspace environment is obtained from `$(VARIANTS)`. The `simv` phony target then depends on the `simv` target in each of the variant directories. Thus, requesting the `simv` target in the unit directory causes each of its selected variants to be built.

Although in this example the variant name is passed as a macro definition via the VCS command line, a more scalable approach is to map the variant name to a set of macro definitions, each specifying an individual feature of the variant, in a central variant header file. This extra level of indirection makes it much easier to maintain the feature set of each variant, as well as to add a new variant.

## 12 Build Caching with Makepp

There is an experimental feature called *build caching* in the development version of Makepp. The purpose of build caching is to store build products in a cache on the filesystem for future use by anyone with access to the filesystem. The cache is indexed with an MD5 hash of all the information that Makepp would use to determine whether an existing generated file is already up-to-date, so this is theoretically as reliable as the incremental build itself.

While this is still a work in progress, it promises to be the definitive alternative to storing generated files in the Perforce server. It provides the advantage of avoiding duplicated build effort, but without the disadvantages of storing generated files, among them:

- It's difficult to know whether the generated file is up-to-date with respect to everything else at the changelist, unless there is a pre-submit trigger that verifies this not only at the submitting client, but also considering any other changelists that have been submitted since the submitting changelist was **p4 sync**'ed.
- Updating (via **p4 sync**) a generated file that depends on a checked-out file will overwrite the differences due to changes in the local workspace. Many build tools (other than Makepp) will not recognize this, and will treat the version from Perforce as being up-to-date.
- Generated files can create a tremendous load on the Perforce server, because they may be large, numerous and frequently changing. (In comparison, there is a limit on the rate at which a person can type.)

If you want to store generated files in Perforce for archival purposes because you don't trust the build system (which is reasonable, because not even Makepp is absolutely reliable), then it is recommended that you do so on a separate server instance, to prevent the archival load from blocking actual development. Using a separate instance effectively precludes the development flow from depending on its contents, but not depending on archives to do normal work is a good thing.

Build caches could also be used for distributing platform-independent files that cannot be generated on every development platform, for example, because the executable that generates them is not available. You'll need to use the `:build_check architecture_independent` rule option to inform Makepp that it's OK to reuse the file across platforms. Because this works only if the desired version was already generated on some other machine, it's a good idea to have a daemon that builds regularly, if not every changelist.

One known issue with build caches is that debug information in object files may contain the absolute path to the source file, and that path may not be accurate if the object file was retrieved from a build cache to a workspace different from the one that generated it. The same problem exists with object files stored in Perforce.

[35] Synopsys

[36] The VCS compile is driven internally by GNU Make. Although the makefiles generated automatically by VCS are much more reliable than a typical GNU makefile, NVIDIA does occasionally encounter issues that need to be worked-around.

## 13 Sandboxing with Makepp

Building NVIDIA's full-chip RTL serially takes a long time (about 4 hours from clean). We accelerate it using distributed Makepp processes with a manual partitioning of the workspace. Makepp has options to detect read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) hazards among concurrent partitions, and we employ those options to ensure that hazards do not lead to nondeterminism in the build products.

We chose this approach over fine-grained parallelism because launching a job via the queuing system (LSF[37]) normally takes between 60 and 90 seconds, and because our NFS implementation does not guarantee that writes are visible over the network until 60 seconds have elapsed.[38] Also, we rely on LSF to reserve software licenses, and not every part of the build uses every licensed tool.

Each partition trusts files in preceding partitions to be up-to-date, because verifying that they are actually up-to-date is time-consuming. Periodic clean building is employed to verify that the preceding partition actually updates the required files. Expert developers also utilize this capability to accelerate their unit-level builds, at their own risk of course.

In theory, Makepp has access to the information that is necessary to do the partitioning automatically, but the implementation of this is far from trivial, and the manually partitioned solution is functioning acceptably to date.

## 14 Testing via the Build System

When you have a reliable incremental build system, you can use it to avoid redundant testing. This basically boils down to storing the result of every test in a file,[39] and running tests by asking the build tool to update the files corresponding to the tests that you want to run.

Although essentially any change to the implementation (other than comment changes, which Makepp can be instructed to ignore) will outdate all the test results, this still has significant advantages. First, it precludes contributors from legitimately neglecting to regress allegedly inconsequential changes because of the time required to run tests, because if the build system determines that the changes are in fact inconsequential, then the regression will run quickly. Second, if submissions to a particular branch are not permitted to overlap one another's regressions (as is the case when using regressed codelines in A4[40]), then the amount of time during which the branch is locked is dramatically reduced if the regression has already been run.

Unfortunately, these advantages rely not only on the incremental build being reliable, but also on the *null build* (the build when all the targets are already up-to-date) being fast. In a hardware development environment, a null build is actually a lot of work. Simply `stat`'ing all of the design files in NVIDIA's workspace takes 2 to 3 minutes, and there is quite a bit of overhead involved in a null build using Makepp. In NVIDIA's null build, the typical CPU utilization on an otherwise unloaded machine is in the neighborhood of 50%, so clearly the execution speed of Perl (as opposed to C, for example) is not so much a limiting factor as the filesystem performance. It's difficult to imagine improving upon this by a large factor without somehow hooking into the filesystem to detect changes as they occur. Still, a null build of many minutes is much faster than a regression of many hours.

## 15 Transitioning from GNU Make to Makepp

Although the syntaxes of Makefiles and Makeppfiles are essentially identical, and their semantics are very similar, Makepp is not really a drop-in replacement for GNU Make. The main reason for this is that GNU Make is so inaccurate that in order to improve its reliability, one has to add strange constructs to the makefile. Such constructs will at best cause Makepp to become inefficient, and at worst will cause it not to work at all. As a result, unless your GNU makefiles are very naïve (which means that you were tolerating a great deal of flakiness), you probably won't be able to reuse them without modification.

This presents a challenge for the transition, because you can't abandon your GNU Make system until the Makepp system is in place and functioning properly, and by default Makepp will try to use an existing Makefile unless a Makeppfile also exists. NVIDIA addressed this challenge by using the **--implicit-load-makeppfile-only**

[37] Platform Computing

[38] Writes are visible within a second at least 99% of the time, but not reliably enough for a build system with thousands of files. Using the close-to-open coherency (CTO) NFS option is supposed to fix this.

[39] For a random test, the filename should contain the random seed, such that the build remains strictly deterministic.

[40] Johnson, op. cit., §2; http://a-4.sourceforge.net/docs/a4intro.html#regression.

Makepp option in its wrapper script. This option causes Makepp to ignore Makefiles, which were still needed in the interim in order to control the GNU Make system. We then created Makeppfiles using a combination of copy-and-paste and selective judgment. When the transition was complete, Makefiles were deleted to avoid the cost of maintaining them in parallel with Makeppfiles.

Well, actually, the reality of the situation is somewhat more complicated than this. NVIDIA has several directories whose builds are still driven by GNU Make, simply because there is no urgent need to transition them. For the most part, this is not a problem because the GNU Make parts of the build happen either strictly before or strictly after the Makepp parts, but there is some potential inefficiency in that the part of the GNU Make build that happens early might build things that the Makepp build won't actually need, and the part that happens late might need things that Makepp didn't build. In a few places, the GNU Make build calls Makepp, which is susceptible to some of the same issues as traditional recursive Make. While these issues are manageable, an ideal build system would be fully self-integrated.

Another issue that NVIDIA experienced during the transition was that naïve contributors tended to copy-and-paste large sections of Makefiles into Makeppfiles without thinking about it, only to be surprised by the subtle (but reasonable and justifiable) semantic differences between them. The syntactic similarity was in this sense a disadvantage.

## 16 Issues and Future Work

Makepp was a relatively immature tool when this effort began, but it has grown up a lot since, in no small part due to its deployment at NVIDIA with over 100 users and tens of thousands of files, not to mention the success of its next generation of Graphics Processing Units (GPU's), relying on it. However, as with any tool, there are still issues and room for improvement.

One of our biggest problems really has nothing to do with Makepp at all, but with the Verilog hardware description language itself. Because macro definitions cross translation units, any change anywhere effectively forces a rescan of everything to determine if any dependencies changed.

A lingering issue is that Makepp (like GNU Make) ignores circular dependencies. This is often the right thing to do, but it's hard to detect when it isn't the right thing to do, and when it's the wrong thing to do, it's hard to figure out what the right thing to do is. Ideally, Makepp would keep rebuilding the same targets repeatedly until they converged, but it would have to give up at some point in case the build isn't going to converge.

Circular dependencies among the loading of various makefiles are also possible, because while loading a first makefile, Makepp might have to load a second makefile in order to build or to find rules for targets in another directory. If the second makefile needs to load the first makefile, then it won't know about any of the first makefile's targets for which a rule has yet to be defined.

While Makepp goes a very long way toward ensuring the accuracy of the build, there are still things that can confuse it, especially allowing for malice. For example if a file is modified without changing its length, and then its timestamp is forged to its previous value, then Makepp won't recognize that the file changed. This can happen accidentally if Makepp is run from within a script which immediately thereafter modifies a generated file in the same time slice. Hiding targets or dependencies from Makepp is still possible, although having scanners for all of the utilized commands makes this much less likely. A truly malicious developer could even defeat MD5.[41]

## 17 Summary

Perforce is a world class source control tool, but there are many configuration management needs that are better met by a good build tool. NVIDIA selected Makepp as its next-generation build tool because of its reliability, extensibility, generality, and the familiarity of its build control language. Makepp has been in successful production use at NVIDIA for over a year.

Variant management is best addressed by the build system, and Makepp is especially amenable to it. Makepp build caching has the potential to obviate the need for storing generated files in Perforce. Regression via incremental build has advantages over traditional exhaustive regression. It is possible to transition from GNU Make to Makepp without breaking the GNU Make system in the meantime.

While no build tool is perfect, Makepp has proved itself sufficiently robust and scalable to build projects of any size reliably and efficiently.

[41] Kaminsky, Dan, "MD5 to Be Considered Harmful Someday."