‘S’ is for ‘Source’: The Role of the Build System in Configuration Management

Anders Johnson, NVIDIA Corporation
Gary Holt, Safe-View Incorporated

What is Configuration Management?

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

In a Nutshell

- Configuration Management is all of the engineering that starts with your hand-edited files and ends with customers using your product

What is Hardware Design?

- Design = Source
- Source ≥ Complexity (Kolmogorov)
- More primitives = More realizable complexity
- More abstraction = Less source

c. 1900 - 1985

- **Paper schematics and hardware prototyping**

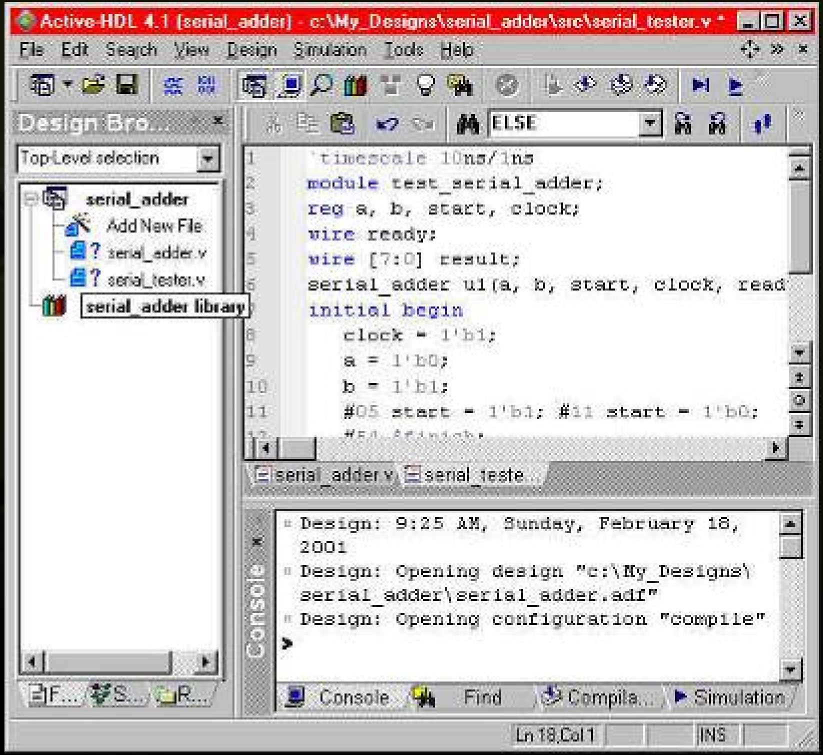
1975 - 1995

- Graphical entry and simulation

1985 - Present

- **Textual entry, simulation and synthesis**

```
module example (/*AUTOARG*/
   // Outputs
   lower_out, o,
   // Inputs
   lower_inb, lower_ina, i
   );
   input i;
   output o;
   /*AUTOINPUT*/
   // Beginning of automatic inputs
   input  lower_ina; // To inst of inst.v
   input  lower_inb; // To inst of inst.v
   // End of automatics
   /*AUTOOUTPUT*/
   // Beginning of automatic output
   output lower_out; // From inst of inst.v
   // End of automatics
   /*AUTOREG*/
   // Beginning of automatic regs
   reg                o;
   // End of automatics
   inst inst (/*AUTOINST*/
              // Outputs
              .lower_out (lower_out),
              // Inputs
              .lower_inb (lower_inb),
              .lower_ina (lower_ina));
   always @ (/*AUTOSENSE*/i) begin
      o = i;
   end
endmodule
```



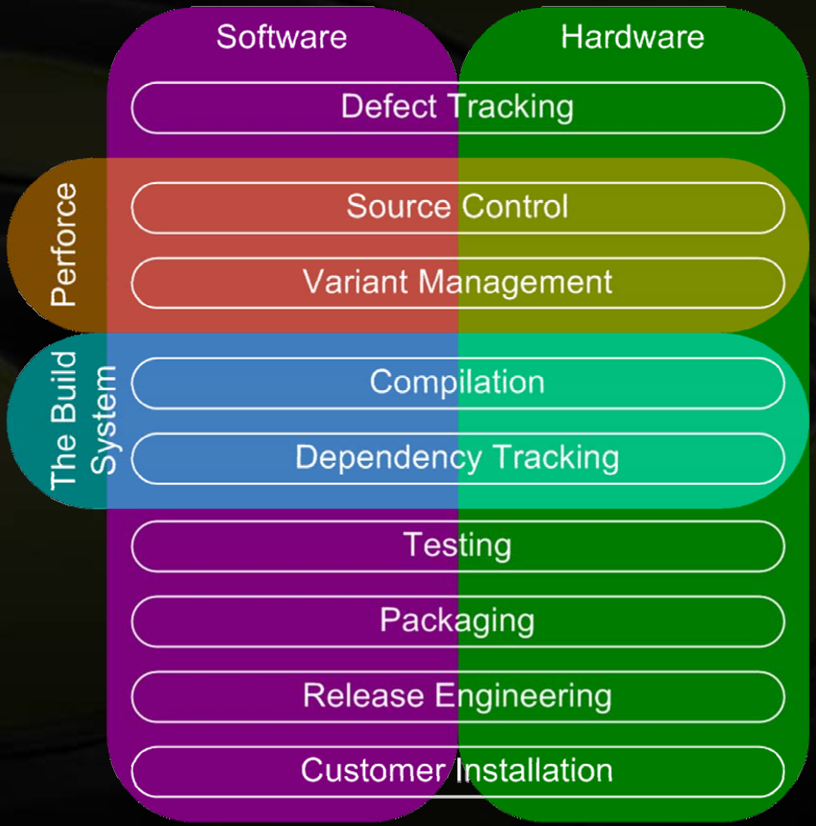
HCM vs. SCM

- Configuration management for a modern, complex hardware design is fundamentally similar to software configuration management
- But…
  - Simulation is slow and sometimes inaccurate
  - Releases take a long time (roughly 3 months to samples)
  - Dependencies are harder to manage, because design is at a lower level of abstraction

What is Perforce?

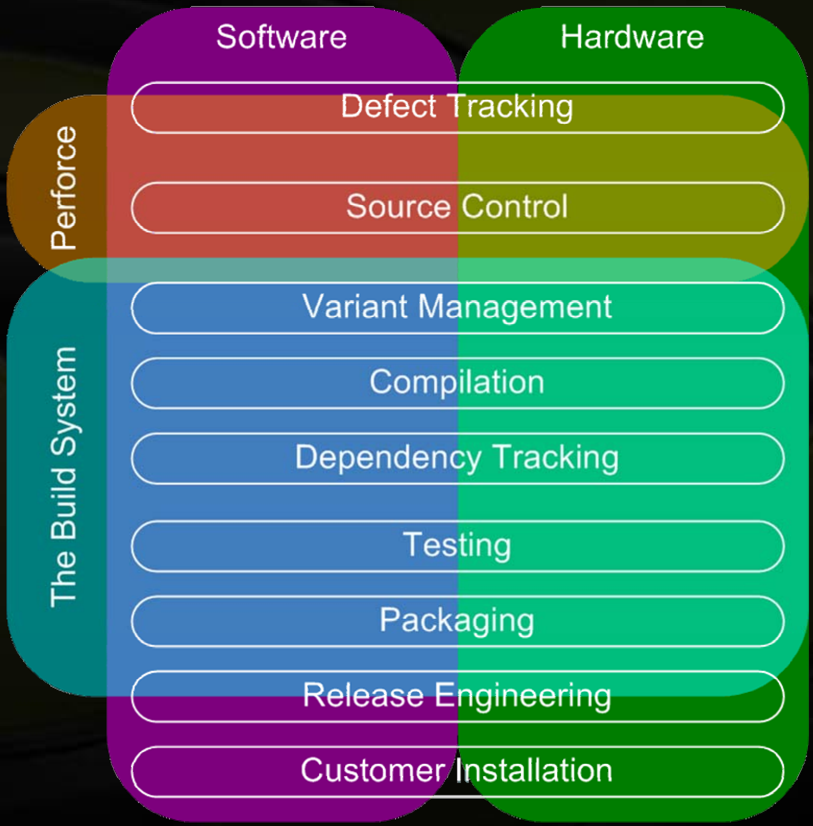
- Perforce is not just for software
- Perforce does not address the entire configuration management problem
- The build (a.k.a. make) system addresses most of what Perforce does not

A Naïve View



- ***Don't use Perforce for variant management***
- **Use Perforce's powerful facility for tracking defects across branches (p4 jobs)**
- **Use the build system to automate testing, packaging and (to some extent) release engineering**

Our View



- **Perforce is *Source* (not Software) configuration management (a.k.a. SCM)**
- **The build system is almost everything else**

Prior Art

- Evidence of this philosophy can be gleaned from the ubiquitous GNU build tool chain:
  - Dependency tracking: `make`, `autoconf`
  - Testing: `make test`
  - Packaging: `make dist`
  - Variant management: `make CFLAGS="-g -DDEBUG"`
- Unfortunately, the GNU build tool chain is limited in its capacity to solve build problems in general

Reliability and Efficiency

- The more you rely on the build system for your mission critical configuration management needs, the more important it is that the build system be reliable and efficient
- Having a reliable, efficient build system is also a plus for routine compilation

Build Problems

Perforce Client Options

- Choose reliability over paranoia
- nomodtime
  - If the build system is relying on the timestamp being updated when the file changes, then this is a must
- clobber
  - If you want sync to succeed when a generated file has become a source file since the previous sync
- rmdir
  - If you want the build to succeed when a new generated file takes the place of an old source directory

Build Reliability Issues

- Missing file dependencies
- Missing implicit file dependencies
- Missing command dependencies
- Missing environment dependencies
- Circular dependencies
- Missing targets
- Using stale generated files
- Using corrupted files
- Using edited generated files
- Recursive make
- “.d” files
- Writing through links
- Wildcards ignoring targets that haven’t been built yet

Do You Feel Lucky?

- If you're very lucky, the unreliable build will fail immediately after it makes a mistake
- If you're lucky, the unreliable build will fail downstream of the first mistake it makes
- If you're unlucky, the unreliable build will succeed, but produce incorrect results!

Missing File Dependencies

- Most common case is missing implicit dependencies
  - For example, if foo.c contains `#include "bar.h"`, then foo.o (*not* foo.c) depends on bar.h
- Consequences:
  - If both the target and the dependency are up-to-date, then it builds nothing, which happens to be the right thing
  - If the dependency is a modified source file, then the target won't get updated before being fed to the linker
  - If the dependency is a generated file that doesn't exist yet, then the compiler uses a version later in the include path
  - If the dependency is an outdated generated file, then the compiler will use it *before* it gets updated

Missing Command Dependencies

- Example:
  - `make`
  - `make CFLAGS="-O3 -DNDEBUG"`
  - GNU Make will *not* recompile in this case!
- Ditto for environment variables, rule actions, compute platform
- Adding dependencies on the makefile is neither necessary nor sufficient
  - Causes all the targets to be rebuilt when only one rule changes
  - Won't catch make include file changes or command-line variable settings changes, etc.

Using Bogus Files

- It is normally bad to use files that…
  - Were once generated, but no longer have a rule
  - Were manually edited after being generated
  - Were left behind by an action that failed
- Common case is the ".o" file left behind by a renamed ".c" file
  - Linker might choose symbols from the stale object file over the symbols from the current object file
- Converse is also a problem: *i.e.* not using targets just because they haven't been built yet

“.d” Files

- From GNU Make texinfo: “Generating Prerequisites Automatically”:

```
%.d: %.c
    @set -e; rm -f $@; \ $(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

- Fails when a header file is deleted
  - Work around by adding a dummy rule for every header
- Doesn’t work for generated headers
  - $(CC) -M doesn’t know where in the include path the header file is going to be found
  - Bizarre failure modes if you also have dummy rules

Build Efficiency Issues

- Recompiling unchanged files
- Unnecessary dependencies (especially makefile dependencies)
- Re-building in the same build run
- Forced rebuilding
- Comparing copy-if-changed targets
- Loading all the makefiles up-front

- *NOTE:* Most of these problems arise from naïve attempts to improve reliability

Copy-if-changed Targets

- A target might not change after it is regenerated, even though some of its dependencies changed
  - No need to continue rippling its effect through the system
- A weak attempt to avoid downstream work:

```
mytarget.tmp: dep1 dep2
    generate_mytarget dep1 dep2 > $@
mytarget: mytarget.tmp
    cmp -s $@ $< || cp $< $@
processed_target: mytarget
    process $< > $@ # takes a long time
```

- In addition to the ugliness, now mytarget always appears out of date, so it gets compared every time

Unnecessary Rebuilding

- Force targets are a bad way to compensate for missing dependencies:

```
.PHONY: FORCE
mytarget: FORCE
    generate_mytarget # depends on lots of things
```

- False dependencies also create unnecessary work
- Recursive make often causes targets to be rebuilt:

```
all: t1 t2
t1: d1
    $(MAKE) -C dir t
    cat dir/t d1 > $@
t2: d2
    $(MAKE) –C dir t
    cat dir/t d2 > $@
```

Idiot-proofing the Build

- If the build system isn't simple to use, then it won't be used correctly
- "make foo" is fine
- The following isn't:
  - make -C path1 all
  - make -C path2 all
  - make foo
  - If you get "unresolved symbol mysym," then "make -C mylib clean," and try again
  - If you get "no rule to make file.h", then "rm *.d" and try again
- Don't require users to do the build system's job!

Makepp

Makepp Overview

- Uses a syntax almost identical to GNU make
- Automatically handles cross-makefile dependencies — no recursive make!
- Finds all include files automatically, and makes them if they don't yet exist (no ".d" files needed)
- Rebuilds if command is different from last build, even if the files haven't changed
- Can ignore stale files
- Can automatically symlink source/object files from another location if they don't exist locally
- Is easily extensible (written in Perl)

Automatic Implicit Dependencies

- Makepp parses shell commands looking for extra dependencies (e.g., `-L` and `-l` options for links)
- Makepp scans source files for #include

  Suppose this rule is used to build `xyz.o`:

  ```
  %.o : %.c
        $(CC) -Idir1 -Idir2 -c $< -o $@
  ```

  1. Recognizes compilation commands from first word(s)
  2. Gets include path from the `-I` options
  3. Scans `xyz.c` for `#include` directives
  4. Finds where each include file is or will be, and makes it a dependency
  5. Applies same process to `#includes` in include files
- Customizable for other shell commands/languages

Multiple directory handling

- Loads all makefiles simultaneously into memory
- Executes commands from different makefiles in the correct order
- Example of ugly of cross-directory dependencies

```
# subdir1/Makeppfile
c : ../subdir2/b
    build_c $<
a :
    build_a
```

```
# subdir2/Makeppfile
b : ../subdir1/a
    build_b $<
```

Makepp executes the following:

```
cd subdir1
build_a
cd ../subdir2
build_b
cd ../subdir1
build_c
```

Build inference

- Makepp computes a list of all files that can be built by all the makefiles it loads (even if the files aren't requested)
  - Makepp starts with the existing files and infers what can be built
  - GNU make starts with final targets and infers how to build them
- Wildcards (e.g., `*.o`) match files that don't yet exist but can be built
- Include files that don't yet exist are made correctly no matter where they are along the include path
- Makepp can generate an automatic "`clean`" target because it knows which files it can build

Implicit Makefile Loading

- If a file in a directory is referenced, makepp will automatically attempt to load a makefile from that directory
- Makefiles do not have to specify which other makefiles are needed — makepp figures it out
- Complete build example:

```
# Top level makefile
our_program: *.o
    $(CC) $^ –Lsubdiraa –Lsubdirbb -laa –lbb –o $@
```

```
# subdiraa/Makefile
libaa.so: *.o
    ld –shared $^ -o $@
```

```
# subdirbb/Makefile
libbb.so: *.o
    ld –shared $^ -o $@
```

Build info files

- Makepp will execute a build command if:
  1. Any file dates have changed since the last build
     - Input file is replaced by an older version
     - Some other program damages the output file
  2. The build command has changed
     - You add `-DDEBUG` to the command line
     - You change from `-g` to `-O2`
  3. The architecture has changed (e.g., from Solaris to Linux)
- Can compare based on checksum of contents
  - Checksum of C source files excludes comments/whitespace so you can re-indent or comment without causing recompilation
- Information about build of abc is stored in `.makepp/abc.mk`
  - You can look back and see what the build command was
  - Can be read by your own scripts

Extensibility

- Makepp is 100% Perl
- Embed Perl code/expressions in your makefile

```
X := $(perl ucfirst($Y))  # Evaluate perl expression
output_dir := .          # Variable is accessible to perl
perl_begin               # Always run snippet of perl code
  -d $output_dir or mkdir $output_dir;
  $file_list = perl_function_to_compute_file_list();
perl_end
# Now $(file_list) contains what the perl code set up
```

- Call your own Perl functions using make syntax

```
X := $(my_special_function arg1, arg2)
```

- New compiler commands or languages can be supported by writing a perl module

Jam/MR and Ant

- Jam/MR and Ant solve some of the same problems, but…
- Makepp solves essentially all of them
- Makepp’s control language is familiar
- Makepp is easy to extend

Makepp at NVIDIA

NVIDIA's Core Business

- NVIDIA builds GPU's (Graphics Processing Units) for rendering cinematic graphics in real time
- Among the most complex integrated circuits on the planet

Variant Management

- Derivative products are crucial for addressing multiple cost vs. performance points with the same basic design
- Maintaining *sustained* variation with inter-file branching is labor-intensive and error-prone
- Maintaining sustained variation with the build system is straightforward
- In the worst case, a former source file can be generated differently depending on the selected variant

Simultaneous Variants

- Multiple products may need to coexist in the same simulation
- Generate each variant in a different location
- Dependencies always refer to a variant location (usually the same variant as the target), so that variant-ness is late-binding
- Makepp is wrappered, so that variant directories can be created during initialization

Repositories for Variant Management

- All source files (including makefiles) are automatically symbolically linked into the variant location when they are needed
- The Missing Link
  - If a dependency is missing, it usually results in a command failing, because makepp won't create the link for it
  - This is a Good Thing

Sandboxing

- NVIDIA uses LSF for distributed processing
- 60-90 seconds of overhead for each process
- File tree is manually partitioned for concurrent makepp processes
- An error results if a process oversteps its sandbox
- Determinism is guaranteed

NVIDIA's Makepp Build Stats

- 17,000 source files, 200MB total
- 10,000 files built by legacy system, 300MB total
- 33,000 files built by Makepp, 2.5GB total
- Top-level compiled simulator target has 4,200 immediate dependencies
- Top-level build is partitioned into 11 phases, with an overall latency of 90 minutes from clean
- 150 users

Perl is Fast Enough

- **NVIDIA's null build spends about 50% of the time in I/O wait, even after optimizing the I/O**
- **Makepp execution latency typically disappears in comparison to the time spent executing build commands**

Future Directions

Build Caching

- Copies of all recently built files are stored on a designated NFS file share
- Indexed with an MD5 of all the dependencies
- *This is the definitive alternative to storing generated data in Perforce*
- Storing generated data in Perforce is evil because…
  - It makes it difficult to maintain coherency with the true source files
  - It can present a Perforce server load that is several orders of magnitude greater than that of true source files

Incremental Testing

- Every test result is a file
  - Result filename must include the random seed, if any
- Running a test is equivalent to updating its results file
- Tests that could not have been affected won't be rerun
- Would you trust *your* build system this much?

In Summary

- Perforce is a *Source* (not Software) Configuration Management tool
- Most of the remainder of the configuration management problem should be addressed with the build system
- Use the build system, not Perforce, for variant management
- Use the build system, not Perforce, for sharing generated files
- Makepp is an exceptionally flexible, scalable, reliable and efficient build tool