IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

Foreword by Lars Bendix, Ph. D., ETP, Lund University, Sweden

Marc Girod **Tatiana Shpichko**

[PACKT] PUBLISHING enterprise professional expertise distilled

Table of Contents

Chapter 3. Build Auditing and Avoidance..... 1

Configuration records..... 1
Validation..... 20
Tying some knots..... 33
Summary..... 39

3
Build Auditing and Avoidance

In this chapter, we'll focus on **derived objects**, and on producing and managing them. We believe that it is what makes *the* competitive advantage of ClearCase, and this makes this chapter, despite its apparent technicality, the most important one. Contrary to the tradition of version control, which still dominates the mindset of most of its competitors, ClearCase allows to *manage directly* what is most valuable to the end user—the products and not the ingredients. This applies to many realms and to the various stages of production: building executables out of *source code* as a particular case, but also, for example, packaging the software, testing, producing documentation, etc. We'll analyze how the user and the tool can collaborate to make management possible and effective.

In this chapter, we are going to cover the following topics:

- Configuration records: how to produce them and to use them; how to optimize makefiles, avoiding various pitfalls, to not only reduce build times, but most importantly, to maximize the stability and the sharing of the derived objects
- Validation: how to use the config records for analysis purposes; how to examine the records of individual derived objects, to reproduce the builds and to assert the quality of the reproduction
- Tying some knots: how to deal with some specific requirements, questioning some "best practices"; we'll finally mention leveraging these techniques to a wider perspective

Configuration records

ClearCase offers to record the software configuration actually used during build transactions. But all records are not equally useful: it depends on the user to help ClearCase to produce records that may become the first source of information concerning the development.

Flat or hierarchical: clearaudit vs. clearmake

To introduce configuration records, the simplest is to first take the **clearaudit** tool. This performs the basic function of recording (**auditing**) the files opened during the hierarchical execution of a process and of the subprocesses directly or indirectly run by it. It does this however only on files located in a vob. Let's thus suppose we are in a view, and under a vob mount directory. We start a clearaudit session, concatenate a couple of files (to `/dev/null`), create one new empty `tag` file, and exit. Then we look at the result as recorded in the **config record** of the `tag` file.

```
$ cd /vob/test/caudit
$ clearaudit
$ cat VobPathConv.pm >/dev/null
$ cat /etc/passwd >/dev/null
$ touch tag
$ exit
$ ct catcr tag
Derived object: /vob/test/caudit/tag@@--05-15T06:19.20613
Target ClearAudit_Shell built by mg.user
Host "sartre" running Linux 2.6.18-128.el5
Reference Time 2010-05-15T06:18:35+01:00, this audit started ############
                                          2010-05-15T06:18:35+01:00
View was vue.fiction.com:/views/mg/mg.vws
Initial working directory was sartre:/vob/test/caudit
----------------------------
MVFS objects:
----------------------------
/vob/perl/lib/site-lib/5.10.0/ClearCase/VobPathConv.pm@@/main/imp/4
<2009-06-21T12:55:49+01:00>
/vob/test/caudit/tag@@--05-15T06:19.20613
```


Comparing to what we got in the *Teaser* chapter, let's note that this case is simpler. In *Teaser*, we did produce a file, `all.tests`, to which the config record got attached; we practiced using the `ct catcr` command. Now again, we need to create one such file, unless our rule modifies one suitable file (in a vob), in which case the config record already gets attached to it and we do not need an extra `tag` file.

We note that `/etc/passwd` did not get recorded in the `tag` config record. It is not a vob object, so recording it (as we did for the `builtin.mk` makefile component in the *Teaser* chapter) would require an explicit mechanism (in the *Teaser*, it was a special makefile macro, which will look into later).

There is thus no *non-MVFS objects* section, and neither any *Build Script* one. Something more important but less obvious—this record is *flat*. Note that the `VobPathConv.pm` module was not mentioned in the derived object `all.tests` config record we have seen in the *Teaser* chapter; and yet it *was* recorded. This is because the record produced by **clearmake** was actually *hierarchical* (which we did not display), and it is `pathconv.cr`, which was referenced explicitly in the `all.tests` config record. This is the main difference between clearaudit and clearmake config records: there are no hidden parts in this clearaudit record; it is all there, flat.

Maybe you wonder where the config record itself is actually stored? The important point here is that although the `tag` file (even if empty) is, at least at this point, stored in the view storage (it is view private data—or lack thereof), the config record is already public, and therefore stored in the vob database (you can detect its existence with the **countdb** utility, which we'll look at in *Chapter 10*, *Administrative Concerns*).

What matters next for us is to understand that audited files may be thought of as **dependencies**, and to see that this makes even more sense in presence of a multi-version database than of a plain filesystem. We shall leave outside our scope how one might compensate the absence of such a database, by mapping the recorded file names to versions in an external repository, in an attempt to decouple the function of clearmake from ClearCase.

The challenge is to use a full list of dependencies, together with the script used to produce a given derived object, as a means to unequivocally **identify** it, and by identification, to understand the most basic function of Software Configuration Management (which one remembers, was once borne as some kind of accounting). This identification is implicit (provided by the system), and operational (the system may use it efficiently).

Let's note that it sets a few requirements:

- The production must be deterministic and causal, grounded on file objects (random behaviors, or dependence on time are excluded)
- All the dependencies must be themselves identified (in the context of ClearCase, files must be in vobs or explicitly designated, elements must be checked in)
- The execution of the script itself must be atomic (closed over oneself: interactions with other processes will cause problems)

Note that the concept of dependency has a precise meaning in the context of makefiles (where it is sometimes also termed as prerequisite).

Makefiles will also allow us, as mentioned earlier, to produce hierarchical config records. The interest lies in the reuse of components, and in their increased stability: coarse components become dependency bottlenecks. Fine granularity, on the contrary, leads to fewer dependencies and therefore to better stability expectations.

Makefile syntaxes—compatibility modes

Makefiles are an old technology, once again popularized by UNIX. It also has slightly diverged in the various environments, and therefore comes in several variants. clearmake has historically taken three parallel strategies:

- Support a minimum common syntax
- Support a compatibility mode with the local platform syntax (Solaris, HP-UX...)
- Support the syntax of **GNU make**, as this soon became available everywhere

This third strategy has clearly come as the winner, and it is the one we recommend and will use.

GNU make has evolved over the years, and clearmake has followed. It offers the richest set of built-in functions, which allows to avoid spawning sub-shells, and thus: depend on the environment, require extra complexity, and incur performance penalties.

It also supports two flavors of variables: *recursively* or *simply* expanding, with the latter offering again opportunities for simplifications and performance improvements in many cases.

The GNU compatibility mode is also the only one supported on Windows. It is invoked on the command line by using the `-C gnu` option, but may also be driven by an environment variable.

We did *not* use it in our *Teaser*, for simplification, as we would have had to explain the presence of one extra makefile component in the config record.

There is a large body of documentation and of expertise about GNU make. This is a valuable asset. Also, it is wise to retain compatibility with the GNU make tool, that is, to ensure that one is able to build using it. The benefit may be to be able to get a wider assistance while tracing difficult problems, or to have a second tool when doubting the first one. Remember however that whatever the similarities, essential differences remain. Not all advice concerning build systems tuned for GNU make is relevant or even valid when applied to clearmake. One good example is the historical controversy concerning a famous paper by Peter Miller: ***Recursive Make Considered Harmful***. While the problems described in it are real, the proposed solution, monolithic top-down build systems, is not in line with our goal of optimizing **winkin**, which obviously could not be a requirement for the author. We'll leave it to the reader to check that our solutions do address Miller's issues, using a radically different strategy.

A first case with clearmake

Our goal will be to describe ways to build hierarchical subsystems, in a way optimizing the stability of the various components and maximizing winkin. We shall grow this system from a first simple one. We'll take our example in C code, but keep it as trivial as possible**:** we'll restrict our functionality to the traditional *Hello World* greeting.

The most natural way to compose a system from components written in C is by creating and using shared libraries.

So, here is a subsystem producing a shared library, containing a function (quite redundant...) to print a single string. The source code is comprised of one header file, and one source file; `wstr.h`:

```
void wstr(const char* const str);
```

and `wstr.c`:

```
#include <stdio.h>
#include "wstr.h"

void wstr(const char* const str) {
  printf("%s\n", str);
}
```

We are, however, more interested in the Makefile:

```
CCASE_BLD_UMASK = 2
SRC = wstr.c
OBJ = $(SRC:.c=.o)
LIB = libwstr.so
```

```
BIN = /vob/tools/bin
CC = $(BIN)/gcc
LD = $(BIN)/ld
CFLAGS = -Wall -fpic
.MAKEFILES_IN_CONFIG_REC: $(LIB)

all: $(LIB)

nolib \
$(LIB): $(OBJ)
        $(LD) -shared -o $@ $(OBJ) -lc
```

Here, we recognize two sections:

- Macro definitions: a name and a value assigned to it
- Rules: a *target*, followed by *dependencies*, and optionally below, indented with TAB characters, with the rule proper, i.e. with a shell script supposed to produce the target

Using clearmake, one need not explicitly name *static* dependencies to elements (as e.g. to `wstr.h` in the example above), as they will be discovered and checked automatically. On the contrary, dependencies on other derived objects are mandatory. They are termed in ClearCase documentation (notably the excellent ***Guide to Building Software***, available both from the IBM website, but only for version 7.0.0 – at least at the time of writing, and as part of the distribution) **build order dependencies**, and this term describes well their essential function, driving the navigation by the make tool of the dependency tree.

Maintaining this list will be the major focus of our attention, and the main tool to ensure the quality of the produced config records.

The rule script is idiosyncratic to our environment: the `-shared` flag and the explicit use of the `-lc` standard C library, as well as the mandated use of the linker (instead of the main *gcc* driver) would be different in some other environment and with other tools. The `-o` flag to specify the output, as well as the `$@` make macro to name the target are more generic.

Let's note that this makefile relies upon the definition of *implicit* (or *pattern*) rules, needed here to perform the compilation proper, i.e. to produce the object file, `wstr.o`, from the source file, `wstr.c`. One advantage of using the C language is that this rule is standard. It uses the `CC` and `CFLAGS` macros, respectively for the compiler and compilation flags. Some of these definitions could (should) be extracted into common makefile components, to be included by similar subsystems:

- `OBJ`, as it is completely generic, and computed from the value of `SRC`.

- `CC` and `LD`: we are using here the Gnu C compiler and linker. These are found from a vob.
- `CFLAGS` is set here to produce *position independent code*, i.e. is only suitable for objects packaged in shared libraries.
- `.MAKEFILES_IN_CONFIG_REC` is technically a special target, although it behaves more like a macro. We met this already in our *Teaser* makefile (found as code from the publisher web site). We'll devote the next paragraph to it. For the time being, and assuming only one library per makefile and one makefile per directory, one could extract it as well. One could also work around these assumptions, but this presents tradeoffs in terms of simplicity, and we'll not do it here.
- `$(LIB)` is also fully generic, depending only on macros. Note the `nolib` guard, ensuring syntactic correctness of the makefile in the case the macro would not be defined.
- Next, we could decide that all the C files found in our directory are actually meant to be linked to our library. This requires that we move away test and any extra files, which is probably advisable anyway, and will help others to make sense of our code. Assuming this, one might use the `wildcard` GNU make function to yield a generic definition for `SRC` in our standard makefile component.

These extractions would leave us with a local makefile such as:

```
STDDIR = /vob/bld/std
LIB = libwstr.so
include $(STDDIR)/stdlibdefs.mk

all: $(LIB)

include $(STDDIR)/stdrules.mk
```

We would still have to define the path to the standard makefile components, `STDDIR`, and the name of the library produced locally, `LIB`. And we would have to define the default target (the first non-special target lexically met), `all`, which is actually equivalent to its dependencies, as having no rule of its own. This target is the one built when invoking clearmake without arguments.

The purpose of these extractions should be clear, although it is twofold:

- Avoiding accidental differences, such what could happen if defining the same list several times, with its tokens in different order, in the makefile: making sure that any information is only represented once.
- Managing by differences, that is, raising the signal/noise value of the local makefile, so that errors would be easier to spot and to fix; only essential information should be presented, if possible.

Our case is still too simple: something we miss here, because we did not meet this need, is to include directories to resolve precompiler directives. These would typically arise from indirect dependencies upon other subsystems. The header files should be included during the compilation of the local source files, but the related libraries should be linked in once (and only once) in the resulting executable. This is an addition we should make into the `CFLAGS` variable, which we moved now into a shared makefile component. This addition may in fact be a boilerplate, and computed from a `LIBS` macro which is only empty in our first simple case. The only problem is that our value of `CFLAGS` was specific to shared libraries (the position independent code), whereas this generic addition could in fact be more general, i.e. implemented in a `stddefs.mk` component, which would itself be included from our `stdlibdefs.mk` one. The point is that we'd like to *modify* (to append to) the value of the generic `CFLAGS`. This is an indication for switching to the *directly expanded* flavor of make macros, which we alluded to in the paragraph on compatibility mode, since modifying a *recursively expanded* macro leads to an infinite regress.

This yields the following state of the standard makefile components, `stdlibdefs.mk`:

```
include $(STDDIR)/stddefs.mk
CFLAGS := $(CFLAGS) -fpic

.MAKEFILES_IN_CONFIG_REC: $(LIB)
```

`stdpgmdefs.mk`:

```
include $(STDDIR)/stddefs.mk
.MAKEFILES_IN_CONFIG_REC: $(PGM)
```

and `stddefs.mk`:

```
CCASE_BLD_UMASK := 2
BIN := /vob/tools/bin
CC := $(BIN)/gcc
LD := $(BIN)/ld
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
LDIRS := $(patsubst %/,%,$(dir $(LIBS)))
LDIR := $(addprefix -L, $(LDIRS))
LNAM := $(addprefix -l, $(patsubst lib%.so,%,$(notdir $(LIBS))))
LINC := $(addprefix -I, $(LDIRS))
CFLAGS := -Wall $(LINC)
```

Note the assignment syntax: this is now a direct assignment and it is only evaluated once, which allows to use the previous value of the macro without fearing the infinite regress.

We changed all the assignments to be direct, with the consequence that their lexical order now matters.

We used GNU make functions liberally, to build lists of options for the tools, from the same initial values. Note that `LIB` and `LIBS` (which we do not actually use yet) are now the only ones under the responsibility of the user.

And we need the following rules (`stdrules.mk`):

```
nopgm \
$(PGM): $(OBJ) $(LIBS)
        $(CC) -o $@ $(OBJ) $(LDIR) $(LNAM)

nolibs \
$(LIBS):
        @cd $(dir $@); $(MAKE)

nolib \
$(LIB): $(OBJ) $(LIBS)
        $(LD) -shared -o $@ $(OBJ) -lc
```

From this file, we only needed the `$(LIB)` rule so far, to which we just added a dependency to potential libraries it would depend upon. This macro being empty in this context does neither affect our build, nor result in any include flag in the compiler options.

But it is time now to turn to the next step, and to *use* this library we just built, precisely via this mechanism. We shall thus use the two additional rules presented right above.

Recording the makefiles

The `.MAKEFILES_IN_CONFIG_REC` special target does what its name says—recording the makefiles in the config record of the objects named as its dependencies. The makefiles are needed in order to reproduce a build, so that it is convenient to record them. However, every target typically requires only a small subset of the information makefile contains— the rule associated with it, with all the macro expanded. If this rule doesn't match textually the script used for a next invocation, clearmake will assume that a different recipe produces another cake and will run it.

But then why *not* to record them systematically? And why using a special mechanism? The answer is simple: makefiles tend to constitute dependency bottlenecks, that is, to constitute a dependency of each and every of the objects they are meant to build and are therefore very volatile. Makefile might be subject to change every time new objects are added. They share this property with directories, and actually with most other containers (such as archives or packages from where to pull out whole installations).

This is why makefiles deserve a special treatment. Like directories, they will (but only when using this special target) get recorded, but will be ignored for dependency matching purposes. Although ignored, of course as file versions and not as expanded scripts: these will still have to match as described above.

Note that in our examples, we do not record makefiles in all the config records, but only for programs and libraries.

As a side remark, recording makefiles with the built-in rules and definitions (for GNU compatibility mode they are the following files under the ClearCase installation directory `/opt/rational/clearcase`: `etc/gnubuiltin.mk` and `etc/gnubuiltinvars.mk`) in this way gives us a good example of *explicit* recording of files outside any vob—the full paths and timestamps are recorded. Of course, all the other makefiles, found in the vobs, are recorded as well, in the *MVFS objects* section of the config record:

```
$ ct catcr hello
Derived object: /vob/apps/hello/helloc/hello@@--05-23T13:15.178584

----------------------------
MVFS objects:
----------------------------
/vob/apps/hello/helloc/Makefile@@/main/3 <2010-05-23T13:15:53+03:00>
/vob/apps/hello/helloc/hello@@--05-23T13:15.178584
/vob/apps/hello/helloc/hello.o@@--05-23T13:15.178583
/vob/bld/std/stddefs.mk@@/main/1 <2010-05-21T01:19:15+03:00>
/vob/bld/std/stdpgmdefs.mk@@/main/1 <2010-05-23T13:10:09+03:00>
/vob/bld/std/stdrules.mk@@/main/1 <2010-05-21T01:19:17+03:00>
/vob/apps/hello/wstr/libwstr.so@@--05-23T12:39.178560
----------------------------
non-MVFS objects:
----------------------------
/opt/rational/clearcase/etc/gnubuiltin.mk <2008-03-07T02:26:59+02:00>
/opt/rational/clearcase/etc/gnubuiltinvars.mk <1998-12-22T02:47:15+02:00>
```

Using remote subsystems

Let's note that our library does not depend so far upon its client code (`hello.c`), and it is good so! It is purely offered. This is achieved by not defining the library and the client code build together in, for example, some common bulky "project" makefile. We shall thus use it from a different, remote directory. We should, in fact, keep the dependency as it is—one way—and avoid creating a top-down architecture that would preclude later reuse of valuable components in other contexts than the ones in which they were initially developed.

Our code will still be trivial, `hello.c`:

```
#include "wstr.h"

int main() {
 wstr("Hello World");
 return 0;
}
```

Again, we'll focus our interest onto the `Makefile`:

```
PGM := hello
LIBS := /vob/apps/hello/wstr/libwstr.so
STDDIR := /vob/bld/std
include $(STDDIR)/stdpgmdefs.mk

all: $(PGM)

include $(STDDIR)/stdrules.mk
```

This should strike as very similar to the former, partly thanks to the extraction of shared makefile components.

We note the use of full and not relative paths. This answers the concern of avoiding to propagate a dependency on the directory of invocation, into the name of the invoked target—being fully qualified, the target name will be the same, independently from where it is being invoked. This concern anticipates on reuse from other possible directories.

Next, we'll notice our using the same `stdrules.mk` file as previously, albeit in it, the two rules we did not use so far.

The `$(PGM)` rule in itself is simple—it links its arguments into an output named as the target (using the `-o` option and the `$@` macro, in the same way as our earlier `$(LIB)` rule). It uses now our `$(LDIR)` and `$(LNAM)` macros. These macros act in a similar way upon `$(LIBS)`, from which they extract lists of directories and filenames respectively, that they format then into options as required by the linker syntax, prefixing the directories with `-L` and the short library identifiers with `-l`.

Its dependency list is what interests us now: or more precisely, not the `$(OBJ)` part, which is again common with the previous case, but the `$(LIBS)` one. `$(LIBS)` is a list , every item of which is produced by our second rule. It is reduced in our example to just one item: `/vob/apps/hello/wstr`. Every call of the rule performs a remote (also named *recursive*) build invocation, i.e. it switches directory to the parent of a library and explicitly invokes `$(MAKE)`.

This will, in fact, result in using our previous case! Here is the transcript:

```
$ clearmake -C gnu
 /vob/tools/bin/gcc -Wall -I/vob/apps/hello/wstr -c hello.c -o hello.o

clearmake[1]: Entering directory `/vob/apps/hello/wstr'
 /vob/tools/bin/gcc -Wall -fpic -c wstr.c -o wstr.o

 /vob/tools/bin/ld -shared -o libwstr.so wstr.o -lc

clearmake[1]: Leaving directory `/vob/apps/hello/wstr'

 /vob/tools/bin/gcc -o hello hello.o -L/vob/apps/hello/wstr -lwstr
```

So, great satisfaction: we made it! Our macros expanded as expected, and our result (hello) works according to specification (note that we have to tell the system where to look for the shared library, using the standard shell macro `LD_LIBRARY_PATH`, when executing it):

```
$ export LD_LIBRARY_PATH=/vob/apps/hello/wstr
$ ./hello
Hello World
```

But, let's try to build again, now with the verbose flag:

```
$ clearmake -C gnu -v
Cannot reuse "/vob/apps/hello/wstr/libwstr.so" - build script mismatch
<<< current build script
>>> "/vob/apps/hello/wstr/libwstr.so" build script
***************
1c1
< /vob/tools/bin/ld -shared -o libwstr.so wstr.o -lc
---
> @cd /vob/apps/hello/wstr/; clearmake
***************

======== Rebuilding "/vob/apps/hello/wstr/libwstr.so" ========
clearmake[1]: Entering directory `/vob/apps/hello/wstr'
`all' is up to date.
clearmake[1]: Leaving directory `/vob/apps/hello/wstr'
============================================================

`all' is up to date.
```

We built the library, but had to rebuild it the next time!? This was not the intention, as nothing has changed that would justify rebuilding. Remember that this may be costly, but also ruins our identification goals.

What is the problem? The transcript tells it very clearly: we have two different rules to produce the `libwstr.so` library—the local one and the remote one. Both concern the same target name.

Remote dependencies

A first idea would be to discriminate between the two target names—the local one (the library name), and the remote one which could be the same with a postfix, e.g. `_build`. The `$(LIBS:=_build)` rule would, however, become a *pseudotarget* in the `stdrules.mk` makefile:

```
nopgm \
$(PGM): $(OBJ) $(LIBS:=_build)
        $(CC) -o $@ $(OBJ) $(LDIR) $(LNAM)

nolibs \
$(LIBS:=_build):
        @cd $(dir $@); $(MAKE)

nolib \
$(LIB): $(OBJ) $(LIBS:=_build)
        $(LD) -shared -o $@ $(OBJ) -lc
```

And it would force the *actual* rebuild of any target depending upon it, with the following message in the verbose build output:

```
Must rebuild "hello" - due to rebuild of subtarget ######################
                                  "/vob/apps/hello/wstr/libwstr.so_build"

$ cd helloc
$ clearmake -C gnu -v
...
Must rebuild "hello" - due to rebuild of subtarget ######################
                                  "/vob/apps/hello/wstr/libwstr.so_build"

======== Rebuilding "hello" ========
 gcc -o hello hello.o -L/vob/apps/hello/wstr -lwstr
Will store derived object "/vob/apps/hello/helloc/hello"
============================================================
```

The cure might seem to be to `touch` such a file (`libwstr.so_build`) at the end of the remote rule (`$(LIBS:=_build)`), but this wouldn't work either. This is because now the target would not itself have **build order dependencies**, and thus, in effect, hide any changes to the library `libwstr.so`, i.e. such changes that would *not* cause the rebuild of the `hello` application.

This appeals thus for a more radical fix: using a layer of pseudotargets, avoiding to depend upon them, but carefully collecting the resulting records.

By "collecting the records", we mean opening the files to which records are attached in the context of a rule and producing a new *tag* file.

The first part, we'll do by reading one line (a minimal amount, maybe not so small in the case of binaries; GNU head has an option to read one byte) with the `head` utility, sending it to `/dev/null`. We shall make a `STAT` macro for this purpose. The second part, we'll do by removing and then touching a tag file (in order to avoid carrying forward dependencies of the previous version).

The changes affect all the shared makefile components. In fact, since we can concentrate the config recording to tag files, and decide to have one per directory (for simplification), with a standard name of `build.tag`, we can move the "special target" (`build.tag`) together with the other targets to `stdrules.mk` and get rid completely of `stdpgmdefs.mk`. The new makefile components are now `stddefs.mk`, `stdrules.mk` and `stdlibdefs.mk`.

`stddefs.mk`:

```
CCASE_BLD_UMASK := 2
BIN := /vob/tools/bin
CC := $(BIN)/gcc
LD := $(BIN)/ld
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
LDIRS := $(patsubst %/, %, $(dir $(LIBS)))
LDIR := $(addprefix -L, $(LDIRS))
LNAM := $(addprefix -l, $(patsubst lib%.so, %, $(notdir $(LIBS))))
LINC := $(addprefix -I, $(LDIRS))
LTAGS := $(addsuffix build.tag, $(dir $(LIBS)))
CFLAGS := -Wall $(LINC)
STAT := head -1 > /dev/null
RECTAG := rm -f build.tag; touch build.tag
```

The new `$(LTAGS)` macro lists the `build.tag` files in all the `$(LIBS)` directories.

`stdrules.mk`:

```
.MAKEFILES_IN_CONFIG_REC: build.tag

build_pgm: $(LIBS:=_build)
 @$(MAKE) TGT=$(PGM) build.tag

build_lib: $(LIBS:=_build)
 @$(MAKE) TGT=$(LIB) build.tag

nolibs \
$(LIBS:=_build):
 @cd $(dir $@); $(MAKE) build_lib

nopgm \
$(PGM): $(OBJ)
 $(CC) -o $(PGM) $(OBJ) $(LDIR) $(LNAM)

build.tag: $(TGT)
 @if [ -z "$(TGT)" ]; then $(MAKE) -f $(MAKEFILE); fi; \
 $(STAT) $(TGT) $(LTAGS); \
 $(RECTAG)

nolib \
$(LIB): $(OBJ)
 $(LD) -shared -o $@ $(OBJ) -lc
```

We set `$(TGT)` on the invocation command lines in order to parameterize the generic `build.tag` target.

`stdlibdefs.mk`:

```
include $(STDDIR)/stddefs.mk
CFLAGS := $(CFLAGS) -fpic
```

The `helloc/Makefile` gets slightly modified accordingly (`$PGM` is replaced by `build_pgm` as the `all` target dependency, i.e. in fact as its alias).

`helloc/Makefile`:

```
PGM := hello
LIBS := /vob/apps/hello/wstr/libwstr.so
STDDIR := /vob/bld/std
include $(STDDIR)/stddefs.mk

all: build_pgm

include $(STDDIR)/stdrules.mk
```

This solves the problem at hand—the `build.tag` files get reevaluated every time (but actually even they are either wink or stay untouched). But the real derived objects are stable—the actual rebuild involving the compiler never gets executed unnecessarily:

```
$ clearmake -C gnu -v
No candidate in current view for "/vob/apps/hello/wstr/libwstr.so_build"

======== Rebuilding "/vob/apps/hello/wstr/libwstr.so_build" ========
clearmake[1]: Entering directory `/vob/apps/hello/wstr'
No candidate in current view for "build_lib"

======== Rebuilding "build_lib" ========
clearmake[2]: Entering directory `/vob/apps/hello/wstr'
`build.tag' is up to date.
clearmake[2]: Leaving directory `/vob/apps/hello/wstr'
========================================================

clearmake[1]: Leaving directory `/vob/apps/hello/wstr'
========================================================

Must rebuild "build_pgm" - due to rebuild of subtarget ##################
                                  "/vob/apps/hello/wstr/libwstr.so_build"

======== Rebuilding "build_pgm" ========
clearmake[1]: Entering directory `/vob/apps/hello/helloc'
`build.tag' is up to date.
clearmake[1]: Leaving directory `/vob/apps/hello/helloc'
========================================================
```

Yet the actual rebuild happens in case any of the dependencies change; however, the situation is not satisfactory as of now.

Multiple evaluation of dependencies

We need to make our case one step more complex yet, by adding a second library/subsystem. The function we add now prints a string in reverse order, and as it happens, depends itself on out first library.
Here is the source code, `srev.h` and `srev.c`.

`srev/srev.h`:

```
char* srev(const char* const s);
void wrev(const char* const str);
```

srev/srev.c:

```
#include <stdlib.h>
#include <string.h>
#include <wstr.h>
#include "srev.h"

char* srev(const char* const s) {
  int i = 0;
  int j = strlen(s);
  char* r = malloc(j);
  while (j) {
   r[i++] = s[--j];
  }
  r[i] = 0;
  return r;
}

void wrev(const char* const str) {
  char* p = srev(str);
  wstr(p);
  free((int*)p);
}
```

and the makefile:

```
# srev/Makefile
ROOT := /vob/apps/hello
LIB := libsrev.so
STDDIR := /vob/bld/std
LIBS := $(ROOT)/wstr/libwstr.so
include $(STDDIR)/stdlibdefs.mk

all: build_lib

include $(STDDIR)/stdrules.mk
```

We slightly modify our main function in `hello.c` and its `Makefile` to use the new functionality:

helloc/hello.c:

```
#include "wstr.h"
#include "srev.h"

int main() {
  const char* const s = "Hello World";
  wstr(s);
  wrev(s);
  return 0;
}
```

helloc/Makefile:

```
PGM := hello
ROOT := /vob/apps/hello
STDDIR := /vob/bld/std
LIBS := $(ROOT)/wstr/libwstr.so $(ROOT)/srev/libsrev.so

include $(STDDIR)/stddefs.mk

all: build_pgm

include $(STDDIR)/stdrules.mk
```

We build successfully, using the unchanged shared makefile components, add the new path to the `LD_LIBRARY_PATH` environment variable and run the application:

```
$ ./hello
Hello World
dlroW olleH
```

It looks right. But nevertheless, there is something wrong.
What is the problem? Let's attempt a rebuild:

```
$ clearmake -C gnu
clearmake[1]: Entering directory `/vob/apps/hello/wstr'
clearmake[2]: Entering directory `/vob/apps/hello/wstr'
`build.tag' is up to date.
clearmake[2]: Leaving directory `/vob/apps/hello/wstr'

clearmake[1]: Leaving directory `/vob/apps/hello/wstr'

clearmake[1]: Entering directory `/vob/apps/hello/srev'
clearmake[2]: Entering directory `/vob/apps/hello/wstr'
clearmake[3]: Entering directory `/vob/apps/hello/wstr'
`build.tag' is up to date.
clearmake[3]: Leaving directory `/vob/apps/hello/wstr'

clearmake[2]: Leaving directory `/vob/apps/hello/wstr'

clearmake[2]: Entering directory `/vob/apps/hello/srev'
`build.tag' is up to date.
clearmake[2]: Leaving directory `/vob/apps/hello/srev'

clearmake[1]: Leaving directory `/vob/apps/hello/srev'

clearmake[1]: Entering directory `/vob/apps/hello/helloc'
`build.tag' is up to date.
clearmake[1]: Leaving directory `/vob/apps/hello/helloc'
```

The problem is that we went twice in the `wstr` directory.

Not a big deal? Unfortunately, it is: this is a major nuisance if the system is not as trivial as ours. Complex builds will take significant time just for clearmake to reach the conclusion that they are up to date. The number of evaluations will only grow, and it will grow fast as the number of paths to reach the leaf nodes from the root in the dependency graph. It will actually affect more fine grained designs than coarse grained ones, and thus encourage people to throw management concerns away.

What is important for us is to avoid the trivial solution of hardcoding one particular traversal of the graph, on the ground that it allows some low-level optimizations.

The solution, however, is not particularly difficult on the basis of our system as it is designed so far! We should note that the problem affects only a single target: `$(LIBS:=_build)`. What we want is to avoid invoking its rule a second time, if it has been invoked once. We must thus keep track of the first time it gets invoked. The natural way is by creating a temporary file. We must do it in a location which doesn't depend on the caller, and in a way which doesn't itself get recorded so that nothing depends on it.

The best location is the simplest one—the directory of the remote target.

The `$(LIBS:=_build)` doesn't itself produce any derived object which would be recorded. So the second requirement is easy to fulfill. We may even add an ounce of optimization by telling clearmake not to record anything, using the `.NO_CONFIG_REC` special target.

The next question is the name of the file: it should be unique, to avoid collisions between possible concurrent builds. The usual trick is to use the process id of the top makefile (the one directly invoked by the user). This *pid* is guaranteed to be unique on the host, so we may add to it the host name, which will protect us against builds running on different machines as well as anticipate on the option of using *distributed builds* (see later). We have to propagate this information through the chain of make invocations. Again, this is standard functionality: we may define this macro in such a way that it gets overridden by the value set in the command line invocation.
The last concern is to clean up these files. We shall do this at the end of a successful build, taking care not to overwrite the possible error code returned in case of failure.

Here are the changes to the two shared makefile components, `stddefs.mk`:

```
BUILD := $(shell hostname).$(shell echo $$$$)
LBLD := $(addsuffix $(BUILD), $(dir $(LIBS)))
```

and `stdrules.mk`:

```
.NO_CONFIG_REC: $(LBLD)
...
build_pgm: $(LIBS:=_build)
         @$(MAKE) BUILD=$(BUILD) TGT=$(PGM) build.tag \
         && rm $(LBLD)

build_lib: $(LIBS:=_build)
         @$(MAKE) BUILD=$(BUILD) TGT=$(LIB) build.tag

nolibs \
$(LIBS:=_build):
         @if [ -z "$(BUILD)" ]; \
          then echo stddefs.mk not included; exit 1; fi; \
          if [ -r $(dir $@)$(BUILD) ]; then exit 0; \
          else touch $(dir $@)$(BUILD); \
          cd $(dir $@) && $(MAKE) BUILD=$(BUILD) build_lib; \
         fi
```

Validation

The result we reached now is simple (all is relative), and certainly too simple to answer all the requirements one might have. For example, one might criticize the fact that the include directives are set once per directory, and not once per file. Also that we create a shared library per directory, which may result in a large number of them. These critiques are valid, but can be addressed, subject to tradeoffs. For instance, the latter could be handled by making intermediate archive libraries, and building the shared libraries from them.

Another batch of critiques might focus on robustness, and error handling and reporting. For example, how would this system react in presence of cycles in the dependency graph (the same subsystem being referenced at different levels in the recursive traversal)?

This introduces us to the tool: **ct catcr -union -check**.

Error reports and their analysis

The answer is that the build might work, and even the resulting executable might as well. Yet, a real problem is lurking with potential consequences, which should rather be detected early, while debugging strange behaviors in the produced deliverables. The transcript might be (we insert numbers in column 1, so that we may refer to the precise reports in the following):

```
$ ct catcr -union -check build.tag
----------------------------
MVFS objects:
----------------------------
1)
Object has multiple versions:
First seen in target "build.tag"
 7 /vob/apps/hello@@/main/mg/11 <2010-05-18T18:39:03+01>
First seen in target "libwstr.so"
 2 /vob/apps/hello@@/main/mg/10 <2010-05-16T16:09:18+01>
2)
Object has multiple versions:
Object has no checked-in element version:
First seen in target "build.tag"
 3 /vob/apps/hello/wstr/Makefile <2010-05-20T09:30:46+01>
First seen in target "libwstr.so"
 1 /vob/apps/hello/wstr/Makefile@@/main/mg/1 <2010-05-16T14:22:30+01>
3)
Object has multiple versions:
First seen in target "build.tag"
 2 /vob/apps/hello/wstr/libwstr.so@@--05-23T12:39.178560
First seen in target "hello"
 3 /vob/apps/hello/wstr/libwstr.so@@--05-25T19:44.178752
```

This transcript is the result of introducing the following addition to `wstr/Makefile`:

```
ROOT := /vob/apps/hello
LIBS := $(ROOT)/srev/libsrev.so
```

In this case, this addition is a pure fake: defining the `LIBS` macro results in a dependency which the code doesn't require. It will thus be easy to fix.

Let's note that we draw now the benefit of having produced a "top level tag file", to which the config record of the whole build got attached. We would like to highlight again the fact that it is not an artificial *global* top, from which any build would have to be run. Our top level is a bottom-up result, i.e. the opposite to an original intention, a "project" top. The risk is to tie to the payload of every subsystem the context in which it was—often accidentally—developed. Such a price tag may effectively defeat the benefits of sharing and reuse. In other words, the build of a subsystem made in the context of a project should be re-usable as such in the context of any other project.

Let's review the analysis provided by `catcr -check`:

- *Directory elements*. Error 1) in the transcript above is: `Object has multiple versions`. This is typical of dependency cycles—the current build remembers of the previous one and, if some objects are modified between the two, the recorded versions will not match. In the case of `/vob/apps/hello@@/main/mg/11`, it is however a directory, and as we already mentioned it, directories are ignored for dependency matching purposes. What this means is that this precise case is not the symptom of a cycle, but an unavoidable difference. We'd rather skip it, or in fact, have `catcr -check` skip it for us, so that we may focus on critical errors first (by default, `catcr` will not list directory versions, but `catcr -check` will report their differences). This can be done by using an additional option, `-type f`, for skipping directories and checking file objects only. One must add that real problems may, at times, lurk behind such differences between directory versions. clearmake will pay attention to some (removing from a directory an entry name used in the previous build), but not all (e.g. adding an entry name which would preempt one used previously if the build was run again).
- *Makefile elements*. The next report 2) is a double one: `Object has multiple versions` and `Object has no checked-in element version`. This is a special case of the previous kind, now concerning a file element. We built before checking in (perfectly normal), but this older build event was not overwritten. Referencing a checked out version usually prevents build reproduction (other views have no access to the data, as it is private to our view). This being an error depends upon the state of our build: it tells us that this build is not shareable. `Makefile` is however a makefile. Our case is thus similar to the previous one with directories—makefiles are ignored for dependency matching purposes, and may be found in multiple versions (as builds using a previous version might not have been invalidated). The unfortunate aspect is that `catcr` does not provide us with a convenient flag to ignore these (usually spurious) reports.
- *Derived objects*. At last, error 3) shows us a real problem: `Object has multiple versions`. This is a derived object this time. So, for some reason this derived object was invalidated and rebuilt, but only in a partial context; the rest of our system still depends on the previous version of this derived object and has not been rebuilt accordingly if there was a real need. We must thus investigate where the problem is.

To investigate case 3), we will use a handy tool, `grepcr`, performing a recursive search in the config records, and thus providing invaluable help for locating cycles in the makefiles. The `grepcr` prints derived objects hierarchies: those that reference the problematic DO in question, `libwstr.so`:

```
$ grepcr /vob/apps/hello/helloc/build.tag libwstr.so
/vob/apps/hello/srev/build.tag@@--05-25T20:41.178893:
 /vob/apps/hello/wstr/build.tag@@--05-25T19:44.178753:
   /vob/apps/hello/srev/build.tag@@--05-25T19:03.178745:
     /vob/apps/hello/wstr/build.tag@@--05-23T14:29.178606:
       /vob/apps/hello/wstr/libwstr.so@@--05-23T12:39.178560:
```

What do we see here? There is clearly a cycle between the `srev` and `wstr` subsystems!

It should be easy now to locate the fake addition to the `wstr/Makefile`, resulting in the cycle and to remove it from there.

There are no other kinds of errors in this transcript, but we already saw that fixing the problem (removing the cycle) will not yield us a fully clean transcript, although a sane situation was restored. This minor *false negative* cannot be avoided:

```
$ ct catcr -union -check -type f build.tag
----------------------------
MVFS objects:
----------------------------
Object has multiple versions:
First seen in target "build.tag"
 7 /vob/apps/hello@@/main/mg/11 <2010-05-18T18:39:03+01>
First seen in target "libwstr.so"
 2 /vob/apps/hello@@/main/mg/10 <2010-05-16T16:09:18+01>
Object has multiple versions:
Object has no checked-in element version:
First seen in target "build.tag"
 3 /vob/apps/hello/wstr/Makefile <2010-05-20T09:30:46+01>
First seen in target "libwstr.so"
 1 /vob/apps/hello/wstr/Makefile@@/main/mg/1 <2010-05-16T14:22:30+01>
```

Here is one more possible error report:

```
Element has multiple names: (OID: ########################################
                                  b966d58d.645d11df.97a2.00:0b:db:7d:45:e7)
 /vob/apps/hello/wstr/stddefs.mk (first seen in libwstr.so)
 /vob/bld/std/stddefs.mk (first seen in build.tag)
```

This occurs in case a file element (`stddefs.mk` in this case) was moved to a different directory. Some derived objects used it from its old location. As we mentioned in our comment of 1), such changes, although only in directories, would force a rebuild, if the object itself was not a makefile. As in the previous case, because the version change did not force a rebuild, the previous version stayed recorded, which results in a spurious error report.

Let's mention that the `catcr -union -check` tool may also be used to assert the consistency of several independently built objects, by specifying multiple arguments. This may be very valuable before attempting an integration.

State of derived objects and reference count

As you produce a derived object, in a standard dynamic view, it is first private, non-shared (but shareable), with a reference count of `1`:

```
$ ct setview view1
$ clearmake
 echo ddd> d.out

$ ct lsdo -l d.out
2010-05-27T16:47:52+03:00 Joe Smith (joe@hostname)
 create derived object "d.out@@--05-27T16:47.179000"
 size of derived object is: 4
 last access: 2010-05-27T16:47:52+03:00
 references: 1 => hostname:/viewstorage/view1.vws
```

If you remove it, the data is lost, but the record remains in the vob database, with a reference count of `0`. You can assess its existence with the `lsdo` tool and the `-zero` flag, or with the countdb one (as already noted while speaking of clearaudit):

```
$ rm d.out
$ ct lsdo -zero -l d.out
2010-05-27T16:47:52+03:00 ???.???
 create derived object "d.out@@--05-27T16:47.179000"
 size of derived object is: 4
 last access: 2010-05-27T16:47:52+03:00
 references: 0
```

Note that such a derived object, which is not **shared**, and with zero references, would always get re-created by a subsequent build (as its data is lost):

```
$ clearmake -d
...
No candidate in current view for "d.out"
>>> 16:55:10.913 (clearmake): Shopping for DO named "d.out" in ##########
                                                VOB directory "/vob/test/tmp@@"
```

```
>>> 16:55:10.943 (clearmake): Removed 0-ref, no-data heap derived #######
                                        object "d.out@@--05-27T16:54.179001"

======== Rebuilding "d.out" ========
 echo ddd> d.out
```

If there exists one reference, and a second view attempts to produce the same derived object, in a context in which identical same dependencies get selected, the derived object **winks in** the new view. As a matter of fact, it first gets **promoted** to the **shared** status, its data gets copied to the vob derived object pool, and the new view gets a reference to it.

Examining the object (from either view) with the `lsdo` tool, we can now see that its status is *shared*, and its reference count `2`:

```
$ ct setview view2
$ clearmake
Wink in derived object "b.out"
$ ct lsdo -l b.out
2010-05-27T16:32:59+03:00 Joe Smith (joe@hostname)
 create derived object "b.out@@--05-27T16:32.178992"
 size of derived object is: 0
 last access: 2010-05-27T16:32:59+03:00
 references: 2 (shared)
 => hostname:/viewstorage/view1.vws
 => hostname:/viewstorage/view2.vws
```

Note that we achieve the same result by using the `winkin` command instead of building. If we first promote a derived object using the `winkin` command, and remove it from the view, only then its data is not lost as it is already stored in the vob derived object pool. Such a derived object, despite its null reference count, may still wink in to other views and thus avoid getting rebuilt.

Removing derived objects

Derived objects get automatically scrubbed periodically once they are not referenced anymore. The details may be tuned using the *scheduler*; we'll pay it some attention in *Chapter 10*, *Administrative Concerns*.

There are cases when you might want to remove some DOs. But as seen previously, until they have been scrubbed, a null reference count doesn't prevent them from winking in back to life. What you might thus want to do, after the reference count has dropped to 0, is to force the scrubbing. There is however a simpler tool than the *scrubber* (used by scheduled jobs) itself: `rmdo`. However, one should only use `rmdo` once the DO has been removed from all the views referencing it, and its reference count has therefore dropped to 0. This requirement is not easy to meet in practice,

when the views of many users are involved. Bypassing it will result in *internal error* reports to the view owners and in the logs.

One reason for removing derived objects could be to reach a situation where `catcr -union -check` reports no errors. This is an optimal baseline to be ready to detect errors easily, if they get introduced.

There are scenarios in which this optimal situation may become spuriously disturbed—by producing equivalent derived objects. This may happen if two concurrent builds are started at nearly the same time. In such a case, neither has yet produced the results, which the other could have winked in. A good design will make such cases of race conditions infrequent, but one cannot avoid them completely. The most common way to produce this adverse situation is by releasing a source code baseline without providing a build to be winked in. We'll see in below in the *Litmus Test* paragraph, how the release procedure may avoid this pitfall. Another way to produce identical DOs is to use *express builds* (i.e. views producing non-shareable derived objects), or some options of clearmake such as `-V` (view only) or `-u` (unconditional), and then to convert the DOs to shareable status (with `winkin` or `view_scrubber -p`, or by checking them in—see later). Such practices are misguided: they give a short term benefit, for a long term (higher level) penalty.

The problem with equivalent DOs is that they introduce spurious differences between config records using different instances out of sets of equivalent ones, and thus pollute the derived object pool. These differences are hard to tell apart, and may thus hide real problems. There is an unfortunate long term bug in clearmake that leverages this issue: when **shopping for derived objects**, clearmake will prefer the *newer* of two equivalent derived objects, therefore propagating a locally created instability in a viral effect. The version of a derived object present in a given view will be validated, but other views will use the newer copy. The result is that the combined system will report differences that will not get cleaned up automatically—they require careful human intervention (to remove the offending duplicate DOs, as explained earlier), or a radical change in basic dependencies.

Here is an illustration of the equivalent derived objects bug:

```
$ ct setview view1
$ ct lsdo wstr.o
--05-23T12:39 "wstr.o@@--05-23T12:39.178559"

$ ct setview view2
$ cd /vob/apps/hello/wstr
$ clearmake -C gnu -u
 /vob/tools/bin/gcc -Wall -fpic -c wstr.c -o wstr.o

 /vob/tools/bin/ld -shared -o libwstr.so wstr.o -lc
```

```
$ ct lsdo wstr.o
--05-27T14:44 "wstr.o@@--05-27T14:44.178969"
--05-23T12:39 "wstr.o@@--05-23T12:39.178559"
$ ct ls wstr.o
wstr.o@@--05-27T14:44.178969

$ ct setview view3
$ cd /vob/apps/hello/wstr
$ clearmake -C gnu -d
...
No candidate in current view for "wstr.o"
>>> 15:27:46.791 (clearmake): Shopping for DO named "wstr.o" in VOB #####
                                        directory "/vob/cifdoc/test/wstr@@"
>>> 15:27:46.798 (clearmake): Evaluating heap derived object ##############
                                              "wstr.o@@--05-27T14:44.178969"
Wink in derived object "wstr.o@@--05-27T14:44.178969"
...
$ ct ls wstr.o
wstr.o@@--05-27T14:44.178969
```

As we can see, in this case when the equivalent derived object has been explicitly created with a `clearmake -u` command, clearmake chooses the "heap derived object", that is, the newest from the two available ones.

Dependencies on the environment and on tools

At this point, we found that the excellent `catcr -union -check` tool has some possible drawbacks—it may report *false negatives* that it is not trivial to get rid of, or even to interpret safely. At the very least, it requires some work and attention.

The unfortunate truth is that it may also err on the other side, and fail to report some problems that could prevent the reproduction of recorded events. A typical cause of such hidden dependencies is the user environment. This is a difficult question as clearmake cannot determine what variables affect where. What is possible is to make sure that environment variables are explicitly set in the makefiles, so that builds get protected from user settings (and that the user is free to set her environment as she pleases). The problem is to determine what variables to define, and for this, we know nothing better than heuristics, and trial and error. So, as soon as an impacting variable has been spotted, define it away—the only shame is to hit the same pitfall a second time.

One particular environment variable is of course `PATH`, which determines the algorithm used to find tools that would not be defined with their full path. It is a good idea to set `PATH` in the makefiles, and a better one is to define the full path of one's tools (define a macro for every tool). One might think that defining `PATH` is sufficient, but it leaves the door open to finding variants of the same tools in different places on different hosts.

Of course, a full path is not enough to determine tool, or its version. One might be tempted to hardcode the version of the tools in their name or their path. This is usually a poor practice, and tends to prevent changes, as the names often spread to several places— scripts, makefiles, documentation, symbolic links. There is no syntax for aliasing such inconvenient names (hence performing the reverse task for one's convenience) that would suit all the contexts of use.

Another easy solution may seem to be accessing the different versions of tools using symbolic links. Symbolic links deserve a special note. They tend to be overused by people without SCM background. One major problem with symbolic links is that they are not a object of their own (apart in one special corner case related to the `lost+found` directory of replicated vobs)—one cannot version them. They are data of directory objects, and as we saw already, directories are ignored for DO matching purposes, except in some limited cases. In addition, symbolic links introduce potentialities for spurious differences in recorded build scripts. Only for these reasons, one should avoid them.

If one wants to discriminate tools, one generic way is to store some specific data (e.g. cksum or version output) about them in makefile macros, and to use these macros in the build rules (even artificially, e.g. by echoing their value to `/dev/null`).

By far the best solution to record tools is however to maintain them in vobs, under common names that would not be tied to a particular version. We'll devote *Chapter 8*, *Tool Maintenance*, to this issue.

Reproducing the build

Fortunately, there is an easy way to make sure one's build is reproducible, and it is to try.

To do it, the first task is to complete the packaging, by applying a convenient label, which will serve as handle (be this our introduction to labels, although we'll be back on their subject in *Chapter 6*, *Primary Metadata*).

First, we need to create label types in every vob referenced. We'll use the config record to tell us which vobs:

```
$ ct catcr -type d -s -flat build.tag | \
perl -nle 'print"mklbtype -nc TYPE\@$_" if s:/\.@@.*$::' | cleartool
```

This command *flattens* the config record, retaining only directory versions. The list is piped to the perl command line invocation that filters vobs root directories only (only they have a dot in their version extended representation: `/vob/tag/.@@/version`) and applies a simple pattern replacement for striping the version information. Using the result, it prints commands to create the label type in every vob, and the output is piped again to cleartool for execution.

This only assumes we attach no comment to the new `TYPE`. This way, we shall create one label type per vob, and these will all have the same (and each of them is the vob local) name. There would be an option to link these types together, using so-called **hyperlinks**, and thus to make them possibly global. This would, however, open a discussion, which we'll defer until *Chapter 9*, *Secondary Metadata*.

What remains now is to apply the label:

```
$ ct mklabel -con build.tag TYPE
```

This command will suffice to traverse all the vobs. It will give us errors if several different versions of the same elements are met during the labeling process, which, as we already decided, might be acceptable in the case of directories and of makefiles. In such occurrences, the label will go to the *latest* version in case the both versions are on the same branch (which may be trivial and thus safe to decide, or may not be so, depending on the topology of the version tree). This is the best possible choice—the differences have been considered by clearmake.

There is of course one case in which this will not work—if we have intentionally used different versions of some elements in different contexts (either by building separately, or by using hard links and exceptional scoped rules based on the alternate names, as alluded in the previous chapter). In such cases, a label will not do: this has to be on at most one version. But exceptions are exceptions, and it is fine to be reminded of them.

The main issue against labeling is the time it takes, but we believe this should rather be addressed than worked around.

Let's however mention one workaround which relates to our current topic: there is a config spec syntax to allow using a config record directly (that is `-config do-pname` option in the view config spec), instead of, as we propose here, via a label applied using it. This does of course shortcut the application! We lack experience of using this rule in practice, and have always preferred to retain a label type as a concrete handle to reproduce a software configuration.

We do, however, record as an **attribute** (let's leave this aspect to Chapter 9) of the label type, the identity of the config record. This identity should allow a different user to retrieve the config record if it is still available, and otherwise, to produce it again. We deal with the latter concern in the following, using the label type itself. The former supposes that one records both the *full path* (guaranteed to be accessible using the label in one's config spec), and the *id of the derived object* in the label type attribute value. ClearCase provides us indeed with a unique identification for every derived object produced, or actually even with two of them:

```
$ ct des -fmt "%n\n%On\n" build.tag
build.tag@@--05-20T09:37.20925
e463cd5e.63ea11df.946a.00:15:60:04:45:5c
```

The `-fmt` option is a standard way with many cleartool commands (for example, `describe` command, which we'll use heavily) to define the format in which one expects the output. It is especially useful to produce synthetic results on a single line, and therefore suitable for grepping. On the contrary, we used it here to produce two lines: the first with the *DO-ID* of the derived object and the other with its **oid**. The former is obviously specific to derived objects. It is human oriented, a compound of a file name, with a time stamp. This doesn't make it memorizable (a wished property of names) but it makes it understandable and easy to relate to other such names. Its weakness lies as often in its *user friendliness*: the timestamp in it is not guaranteed to be stable. On the contrary, it will change after one year to explicitly mention the year. This is what leads us to store at least in addition, the latter line, the oid.

This is, on the contrary, a low-level **object id**, a kind shared with all ClearCase objects, elements, or types. Note that this is not the lowest possible id: there also exists a seldom met, e.g. in some error reports, *dbid* at the underlying database level.

This oid is stable; however, it is not sufficient alone, and not only from a cognitive point of view: reconstructing the original DO-ID from it is not trivial. One can use the following to retrieve the original DO-ID timestamp part:

```
$ ct des -fmt "%n\n" oid:e463cd5e.63ea11df.946a.00:15:60:04:45:5c
e463cd5e.63ea11df.946a.00:15:60:04:45:5c@@--05-20T09:37.20925
```

And as for the file name part of the DO-ID, we need to keep track of it ourselves, and that is why we record it separately as well:

```
$ ct des -fmt "%Na\n" lbtype:TYPE@/vob/apps | tr ' ' '\n'
ConfigRecordDO="build.tag@@--05-20T09:37.20925"
ConfigRecordOID="e463cd5e.63ea11df.946a.00:15:60:04:45:5c"
```

Now that we have the labels in place, we may want to lock the label types as a proof that nothing could affect them after the time stamp of the locking event.

The label type `TYPE`, which we have just created, attributed, and applied, can now serve two different but related purposes: first, for the build reproducibility (see the quick-check *Litmus test* below), and the second, for fetching the derived object, and the whole software configuration along with it by explicitly winkin from the command line:

```
$ ct winkin -r build.tag@@--05-20T09:37.20925
```

This will be useful for our "customers", to access our proposal with the least effort; but we won't get deeper into that right now.

Litmus test

We can now create a new view, and set a one-line config spec using the label type as its single rule:

```
element * TYPE
```

What we want to do is to build using the same makefile and our target is to produce the DO in the first place. And what we expect to witness is a full winkin and build avoidance.

The point is: if the build *does* build anything, then something went wrong in the recording —we do not have the exact same dependencies.

At this point we do have data to examine; we may run `catcr -union -check` on the two topmost build tags and narrow the differences down using out `grepcr`.

Note that this test, however good, cannot prove we have no hidden dependency on the environment. The winkin is only as good as the record is—a full winkin is not a proof that clearmake could have built what it winked in.

This procedure may seem heavy, but it all depends on the frequency of mismatches, and after a first investment in tuning the makefiles, and there, especially the dependency lists, the whole system will converge to a stable one, with minor incremental issues bound to the latest changes.

This may also serve as the basis for the further enhancements, just as we did illustrate in our *Teaser*.

In that case the label type also helps the new developer to find out, *how* the build was actually made, that is, to find the Makefile along with its location and the target!:

```
$ ct des -fmt "%Na\n" lbtype:TYPE@/vob/apps | tr ' ' '\n'
ConfigRecordDO="build.tag@@--05-20T09:37.20925"
ConfigRecordOID="e463cd5e.63ea11df.946a.00:15:60:04:45:5c"

$ cleartool catcr build.tag@@--05-20T09:37.20925
Derived object: build.tag@@--05-20T09:37.20925
Target build.tag built by joe
Host "tarzan" running Linux 2.6.18-128.el5
Reference Time 2010-04-1T07:51:07Z, this audit started ###################
                                                    2010-04-01T07:51:07Z
View was vue.fiction.com:/views/jane/joe.vws
Initial working directory was /vob/apps/hello/helloc
----------------------------
MVFS objects:
----------------------------
/vob/apps/hello/helloc/build.tag@@--05-29T15:29.179120
/vob/apps/hello/helloc/Makefile@@/main/cif/5 <2010-05-24T19:16:03+03:00>
/vob/apps/hello/srev/build.tag@@--05-29T15:29.179117
/vob/bld/std/stddefs.mk@@/main/cif/3 <2010-05-25T16:58:25+03:00>
/vob/bld/std/stdrules.mk@@/main/cif/8 <2010-05-29T15:25:48+03:00>
/vob/apps/hello/wstr/build.tag@@--05-29T15:28.179108
----------------------------
non-MVFS objects:
----------------------------
/opt/rational/clearcase/etc/gnubuiltin.mk <2008-03-07T02:26:59+02:00>
/opt/rational/clearcase/etc/gnubuiltinvars.mk <1998-12-22T02:47:15+02:00>
----------------------------
Variables and Options:
----------------------------
LTAGS=/vob/apps/hello/wstr/build.tag /vob/apps/hello/srev/build.tag
MAKE=clearmake
MAKEFILE=Makefile
RECTAG=rm -f build.tag; touch build.tag
STAT=head -1 > /dev/null
TGT=
----------------------------
Build Script:
----------------------------
 @if [ -z "" ]; then clearmake -f Makefile all; fi; ####################
                       head -1 > /dev/null /vob/apps/hello/wstr/build.tag
         /vob/apps/hello/srev/build.tag; rm -f build.tag; touch build.tag
----------------------------
```

Then we know we need to use the `Makefile` makefile from the `/vob/apps/hello/helloc` directory (note the line, `Initial working directory was /vob/apps/hello/helloc`, in the transcript above), and that the makefile target is `build.tag` (note the line, `Target build.tag built by joe`, in the transcript).

What is more important is that this ought to describe the situation following a release: any *official* or just *higher level* build using our latest contribution should only *validate* it, i.e. promote it as the baseline status for others. The most convincing way in which it could achieve this is by **building nothing anew**. One obvious condition for this to be possible is that the release procedure itself should provide a build suitable for winkin. The build is a necessary part of the release, and it must take place before updating the baseline.

We'll have to show in Chapter 6 how to avoid breaking this with adverse release processes.

Let's stress here how this opposes most traditional strategies, in which the build manager confidence is grounded in the fact that *she* built everything in *her* controlled build environment. In the context of clearmake, we may ground confidence on **transparency** and **management**, instead of on *opacity* and *control*! We may unleash the benefits of collaboration.

One result we obtained is the identification by the system of the derived objects, as members of a family, which allows them to be elected as a possible, if not the preferred, ClearCase implementation of the concept of configuration item. We cannot avoid to reckon that the tight mapping of the DO family identity to a full path name sounds like an unfortunate overspecification.

We reached here to the logical conclusion of this chapter, which we'll wrap up nevertheless once again before the beginning of the next one.

However, before concluding, we must handle a few issues that did not find a suitable place in our presentation.

Tying some knots

Some issues could not find their natural place in the flow of our presentation, but must be dealt with nevertheless.

Ties between vobs and views

After we have produced shared derived objects, and these have thus been moved to some vobs' derived object pools, we broke the clean separation between vobs and views. We can easily assess this by running the `describe` command with its `-long` flag on some vob object:

```
$ ct des -l vob:.
versioned object base "/vob/apps"
...
 VOB holds objects from the following views:
...
vue.fiction.com:/views/mg/mg.vws [uuid ##################################
                                  d2d16962.837211de.9736.00:01:84:2b:ec:ee]
vue.fiction.com:/views/mg/mg2.vws [uuid ##################################
                                  abcb9897.604e11df.8511.00:01:84:2b:ec:ee]
...
```

This is certainly a minor detail which may be ignored most of the time. It will, however, affect removing of views and resynchronizing them with the vobs after a recovery from backup or another replica. We'll be back to the former in Chapter 10.

Distributed or parallel builds

The issue of builds distribution is *in*: it brings along with load balancing, significant hopes of performance gains and optimal resource utilization. It is supported by clearmake using `-J` options standard to other make tools and to Gnu make in particular (the option is `-j` in Gnu make). There are, however, pros and cons.

Let's set up an environment for distributed builds and collect some data. We need to create a file with a list of hosts in our home directory, with a name starting with `.bldhost` and a suffix matching an environment variable `CCASE_HOST_TYPE` (which would rather be a makefile macro), and to ensure that the remote shell used by clearmake is enabled for our use, between these hosts:

```
$ export CCASE_HOST_TYPE=test
$ cat ~/.bldhost.test
sartre
beauvoir
$ ll /opt/rational/clearcase/etc/rsh
lrwxrwxrwx 1 root other 12 Aug 3 2007 /opt/rational/clearcase/etc/rsh ###
                                                          -> /usr/ucb/rsh
$ cat <<eot> ~/.rhosts
> sartre
> beauvoir
> eot
$ /opt/rational/clearcase/etc/rsh beauvoir echo hello
hello
```

Instead of configuring `rsh`, we could as well have set the `CCASE_ABE_STARTER_PN` environment variable to point to, for example, `ssh`, and used it.

With this setup completed, we are ready to fire the first try of the distributed build, which produces some error though:

```
$ clearmake -C gnu -J 2
Build host status:
 Host sartre unacceptable: only 36% idle, build hosts file requests #####
                                                                  50% idle.
...
```

What is the problem? Well, we need to fine-tune the default 50 percent idleness limit (a lower value doesn't seem unreasonable):

```
$ cat ~/.bldhost.test
-idle 30
sartre
beauvoir
```

Then for the purpose of the test, as explained earlier, we removed all the derived objects, both from all views and, as vob owner with `rmdo`, from the derived object pool.

We ran the following command, thus under a utility measuring the time spent (in user and kernel spaces, then from a *wallclock* perspective: *real*). We slightly skim the output:

```
$ time clearmake -C gnu -J 2
Rebuilding "/vob/apps/hello/wstr/libwstr.so_build" on host "sartre"
Rebuilding "/vob/apps/hello/srev/libsrev.so_build" on host "beauvoir"

======== Finished "/vob/apps/hello/wstr/libwstr.so_build" on host #######
                                                      "sartre" ========
...
======== Finished "/vob/apps/hello/srev/libsrev.so_build" on host #######
                                                    "beauvoir" ========
...
======== Finished "/vob/apps/hello/wstr/libwstr.so_build" on host #######
                                                      "sartre" ========
...
======== Finished "build.tag" on host "sartre" ========
============================================================

clearmake[1]: Leaving directory `/vob/apps/hello/helloc'
============================================================

real 0m5.735s
user 0m0.515s
sys 0m0.995s
```

The similar time test for a non-distributed build fires the following figures:

```
$ time clearmake -C gnu
...
real 0m2.454s
user 0m0.476s
sys 0m1.087s
```

The results (on such a small build system) are quite clear:

- The build passes and produces identical results to the non-distributed one
- There is an overhead to distribute the builds: the shortest times achieved even with the `-J` flag are met when the actual building happens on one single host (because other hosts are temporarily overloaded)
- The overhead is comparatively much larger in subsequent builds (i.e. when the build process is mostly just asserting that nothing needs to be built)

The first point cannot be neglected: a correct build system, with dependencies completely described, should be distributable. As a matter of fact, distributing the build is a sound test of the build system. An incorrect build with insufficient dependencies will fail only randomly, depending on which targets are distributed to different hosts, so that a one-time success is no guarantee.

The second and third points are obvious in afterthought, but must be kept in mind: the overhead will penalize a user building a small system. But it will also penalize all users on average—there is a bounty paid to distributing builds. This bounty may be tolerable if there are computing and networking resources in excess, but it becomes unjustified under heavy load. One might also guess that the overhead penalizes fine-grained systems more than coarse-grained ones, but we do not have facts to back this guess.

A break-even is to be expected for a certain size of the build, which is neither easy to compute nor stable. One could expect that distributing builds may win on building from scratch, but one should not be surprised to see distributed builds compare poorly on incremental builds, which should be the main target of our SCM focus: **manage by differences**.

Let's however admit that parallel building doesn't work well with recursion in clearmake: until version 7.1, the value given by the user via the `-J` flag or the `CCASE_CONC` variable (the maximum number of concurrent builds) is propagated and implicitly reused independently by every recursive clearmake invocation, potentially resulting in an exponential explosion of build jobs; in 7.1, the value is is not propagated, so that only the initial build targets may be distributed. In order to fix this problem, one may consider computing a `JVALUE` parameter, and use it to spawn sub-makes with: `$(MAKE) -J $(JVALUE)`.

The computation will need the original value of `CCASE_CONC` (propagated e.g. as `ORIG_CONC`, since `CCASE_CONC` is locally overridden by the parent `-J` option), as well as a count of the processes currently running, obtained with something similar to `$(shell pgrep clearcase/etc/abe)`. Note that *abe* (audited build executor) processes are launched for build jobs on remote hosts, not for multiple jobs on the same host (if one uses the same host multiple times in one's `bldhost` file, for example, to take advantage of a cluster architecture). In this latter case, one would have to monitor something else than the number of abe processes, maybe just clearmake ones.

Here is a code example, using `wc` (UNIX word count) to count the number of clearmake processes, and `dc` (UNIX desk calculator) to calculate the value of the `JVALUE` parameter:

```
ifeq ($(MAKE), clearmake)
   ORIG_CONC := $(CCASE_CONC)
   LCONC = $(shell pgrep clearmake | wc -l)
   JVALUE = $(shell p=`echo $(ORIG_CONC) $(LCONC) - p | dc`; \
     if [ $$p -lt 0 ]; then echo 0; else echo $$p; fi)
   OPTS = ORIG_CONC=$(ORIG_CONC) -J $(JVALUE)
endif
```

One would use this value explicitly in the remote invocations (only on pseudo targets: beware recording it into the config records!):

```
      $(MAKE) $(OPTS)
```

Note also that such measurements may easily be fooled if started concurrently (thus too early to take each other into account). This is why GNU make actually uses a more robust technology, based on serialized IPC.

In some cases (such as precisely, taking advantage of multi-processor or cluster architectures) such complexity (or better) might be worth the while. Otherwise, understanding these aspects, one may tune the values of `CCASE_CONC` to get some benefit of distribution for one's system.

Let's conclude our critique with the following note: using derived objects already built by others (winking them in), *is* actually a form of distributed build! This is true at least from the point of view of the resource utilization.

Staging

The practice of making elements of one's critical deliverables and checking in new versions, also referred to as *staging*, is often recommended, even in IBM/Rational own documentation.

Creating an element, checking out, and checking in a version, will be the topic of the next chapter.

Doing it inside the build brings in a few additional issues, such as:

- Accessing the checked-in versions in a view in which `LATEST` has been locked at the time of starting the build (as with a `-time` clause in a config spec): this is best done by using a label, which must first be applied, which leads to the next problem.
- Accessing checked-out versions with commands such as `mklabel`: one needs to use an explicit `@@/main/CHECKEDOUT` extension, meaning that one has to record or to compute the branch from where the version was checked out.
- Using the additional `-c` option of clearmake to check out the derived object versions prior to modifying them.

Several reasons may be invoked in favor of staging:

- Ensuring that critical deliverables won't be lost (especially after they have been delivered to customers).
- Providing a performance benefit, by avoiding configuration lookup.
- Sharing derived objects with MultiSite.

The main price is, however, to create alternative dependency trees: ones with opaque nodes. The makefile system stops being transparent.

Of course, the `catcr -union -check` consistency test is still available—checked in derived objects remain derived objects. But asserting the consistency of the build becomes a political issue: it is not collaborative anymore.

Staging is thus not downright incompatible with derived object management under clearmake, but it surely competes against it.

We believe the concerns listed above may be addressed in other ways:

- Shared derived objects won't get scrubbed if they remained referenced. One may easily keep a view with a reference to the top of a config record hierarchy, which will ensure the persistence of the critical assets.
- We believe that we sufficiently addressed the performance issues and showed that the gain of staging is at best only minimal against a well-tuned build system.
- Sharing derived objects across sites is not supported by MultiSite. Sharing the data may be uselessly expensive: network bandwidth has progressed greatly, but latency has not, and is not expected to, under the current laws of physics.

- In any case, even bandwidth has not progressed at the same speed as CPU and clocks. It does make sense to share config records, for comparison purposes, which is possible with a `-cr` option of the `checkin` tool. This option has the same issues as staging proper, with placing references to the derived objects into journaling records (known as *oplogs*) used for the replication. We shall come back to this in *Chapter 5*, *MultiSite Concerns*.

Application to other tasks than mere builds

In the spirit of our *Teaser*, we want to stress that the techniques developed in the chapter may be applied to other tasks than the ones understood with a narrow acceptation of *building*. In general, they may be used to avoid running again and unnecessarily arbitrary tasks, by managing the artifacts these ones produce. An excellent example of a domain of application is test avoidance, and especially regression testing.

Summary

We are aware that we dealt with a mass of technical details. We believe these details were essential to satisfy the extremely high expectations we place on a *full* use of the avoidance and winkin machinery that ClearCase offers us. In the precision with which the dependencies are described lies the scalability and the economy of the recording, and hence eventually, the justification for the strong points we'll make in our conclusion.

We would like however to bring our reader's attention, away from those technical details, to the *paradigm shift* that has taken place: from *producing deliverables*, to **reproducing** *a task* performed elsewhere by somebody else. *Formal sameness* is now detected by the *system* (clearmake), and guaranteed by avoiding to merely produce *again* the same, or similar in some loose sense, deliverables. The benefit is not so much a matter of build speed or of space saving. It lies in the **stability** that could be obtained, and communicated between developers, for the incremental steps of the development process.

This is the insight we'll have to carry with us while exploring the rest of ClearCase.