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

8
Tools Maintenance

Tools are an essential concern in build management. If we consider the mass of information that software development has to deal with, and structure it as sets of software configurations, that is, (using the formidable tool ClearCase offers us) as dependency trees, then we find tools at both boundaries: as the *roots* of the dependencies, with sources; at their *leaves*, with the customer deliverables (as part of the run-time environment).

The opacity of tools, as well as that of sources, is of course only a matter of viewpoint: the boundaries are only surfaces of separation between realms of distinct coherence. Some other organization is responsible for developing and delivering them, but typically under a different SCM.

This chapter will be structured in a traditional way, answering two questions about maintaining tools under ClearCase, that is maintaining them in vobs:

- Why?
- How?

In the first part, we'll offer an analysis, which is too often skipped, that is, the questions are skipped even after the answers have popped up. We'll thus, as earlier, question some traditional answers.

The second part we choose to confront through an example, convinced as we are that *the Devil is in the details*. The review will show the practical difficulty of anticipating changes beyond one's control and the compromises this leads to.

Why?

There are many reasons for managing third-party tools under ClearCase:

- Dependency control during builds: we want to ensure the reproducibility of our build. As the build depends on third-party tools (libraries, compiler, and so on), we naturally want to keep track of those dependencies.
- Safety with updates: we may easily switch back and forth between successive versions.
- Referential transparency: we must be able to switch between versions of tools without having to change our software or makefiles.
- Flexibility: in most cases, simultaneous use of different versions must be possible.
- Localization/Fixing/Enhancing: we can configure, or even correct or modify products of which we have the sources. We could report or publish such modifications, in the hope that they are being integrated into the next official versions.
- Tracking of indirect dependencies: third-party components may themselves use chains of other components, which we may use in different contexts. We want to be able to detect such dependencies and possible conflicts between them.
- Replication via MultiSite: we can share the installed products (and if need be, maintain different local configurations).

Dependency control

We want to conform our results to our intentions, and thus to exclude any factors of instability. We want to be able to compare our results: our own successive ones, as well as ours with others'. We need to ensure the consistency of our builds in that all parts of them use the same or compatible tools. And finally we want to be able to analyse the results in order to identify and fix the cause of problems, which we need to narrow down and reproduce in isolation.

If our build depends on a tool *X*, and *X* is not under ClearCase, then clearmake offers only limited support:

- It records a dependency on the build script, with the values of makefile variables expanded
- It records the time stamps of *declared* dependencies

This seems to open the way to two strategies: explicit declaration and hardcoding version information in path names. The former is not really an option on a larger scale: there are just too many files to name. It is important however to investigate it, as it may pop up as a recourse in case of failures with the other.

We'll thus try to explore both and show their shortcomings, so as to justify the extra effort needed to import and maintain the tools in vobs.

Safety with updates

If the product *X* is not managed in ClearCase:

- Derived objects that depend on different versions of the product *X*, will not be validated properly; unconditional rebuild would be required
- Having upgraded *X*, one would need to keep apart an installation of the previous version, for example, in order to reproduce a bug reported by a customer in the configuration in which it was used

Explicitly declare tools as dependencies?

According to the documentation (*Tracking non-MVFS files in* ***ClearCase IBM Rational ClearCase Guide to Building Software***), clearmake tracks the checksum of explicit dependencies not in a vob.
We found this description insufficient and misleading, as we'll show now.

Let's create a tool named `makefoo`, producing a derived object, `foo`. Let's place it in the `/tmp` directory (hence *not* under ClearCase), and record its time stamp (force it to an arbitrary value) and its size:

```
$ cat /tmp/makefoo
#!/usr/bin/bash

echo zoo > foo
$ touch 08141200 /tmp/makefoo
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc  jgroup 32 Aug 14 12:00 /tmp/makefoo
```

Let's now use our tool in a makefile, in a vob, being careful to declare it as a dependency of our `foo` target:

```
$ cat Makefile
foo: /tmp/makefoo
        /tmp/makefoo
$ clearmake
/tmp/makefoo
```

```
$ ct catcr foo
Derived object: /vob/test/nmvfs/foo@@--08-14T12:07.23206
...
Initial working directory was /vob/test/nmvfs
----------------------------
MVFS objects:
----------------------------
/vob/test/nmvfs/foo@@--08-14T12:07.23206
----------------------------
non-MVFS objects:
----------------------------
/tmp/makefoo <2010-08-14T12:00:00+01>
----------------------------
Build Script:
----------------------------
 /tmp/makefoo
----------------------------
$ clearmake
`foo' is up to date.
```

We checked that a second build gets avoided as it should.

Let's now modify our tool (we replace `zoo` with `bar`), but keeping its size and time stamp unchanged. Of course, its checksum changes:

```
$ perl -pi -e 's%zoo%bar%' /tmp/makefoo
$ touch 08141200 /tmp/makefoo
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc jgroup 32 Aug 14 12:00 /tmp/makefoo
$ clearmake
`foo' is up to date.
```

Let's face it: we are disappointed. From the above referenced documentation, clearmake should have detected the checksum change, and rebuilt. Let's investigate further.

We then force the rebuild (by removing our first product) and check that the tool produces a different result.

```
$ cat foo
zoo
$ rm foo
$ clearmake
 /tmp/makefoo

$ cat foo
bar
```

Our next attempt is now to do a similar modification, only this time, not preserving the size of the tool. We check that the rebuild happens:

```
$ perl -pi -e 's%bar%bar1%' /tmp/makefoo
$ touch 08141200 /tmp/makefoo
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc jgroup 33 Aug 14 12:00 /tmp/makefoo
$ clearmake
 /tmp/makefoo

$ cat foo
bar1
```

Last, we modify only the time stamp—first, to a value newer than the recorded one for the tool, but older than this of the derived object:

```
$ touch 08141210 /tmp/makefoo
$ clearmake
'foo' is up to date.
```

No rebuild! Secondly we set it to a date newer than this of the derived object, which would trigger the behavior of a "standard" make tool (using the `-T` flag would show it, but would obviously defeat our purpose):

```
$ ll foo
-rw-rw-r-- 1 marc jgroup 5 Aug 14 12:13 foo
$ touch /tmp/makefoo
$ clearmake
'foo' is up to date.
$ ll /tmp/makefoo
-rwxrwxr-x 1 marc jgroup 33 Aug 14 12:39 /tmp/makefoo
```

This is downright embarrassing... The explanation for all these behaviors comes from reading the improbable small print of a ***technote (#1386111):*** *both* the time stamp and the checksum must change for clearmake to rebuild! Alone, neither of them is sufficient (but as it happens, the size seems to be...)

We are of course nit-picking (unintentionally!): seldom will the time stamp remain the same while the contents changes. But this possibility, added to the discontinuity with the standard make behavior (ignoring a time stamp newer than this of the source), is enough to cast a doubt: we cannot depend on this functionality.

Lacking confidence in this behavior of the build system, one traditionally resorts to `make clean`, or to building *unconditionally*.

This is not only sub-optimal in terms of build performance, but introduces discontinuities in the development process, and thus trade-offs to upgrading or just tuning one's tools.

Furthermore, it doesn't encourage supporting fine-tuning one's build system, and therefore feeds the vicious circle leading to further lack of confidence.

ClearCase has better to offer!

Let's just move the tool in a vob, as a view private object. We may drop declaring it:

```
$ cat Makefile
foo:
 /vob/test/nmvfs/makefoo
$ ll makefoo
-rwxrwxr-x 1 marc jgroup 33 Aug 14 15:24 makefoo
$ clearmake
 /vob/test/nmvfs/makefoo

$ clearmake
`foo' is up to date.
$ touch makefoo
$ clearmake
 /vob/test/nmvfs/makefoo

$ ct catcr foo | grep 2010
Reference Time 2010-08-14T15:34:12+01, this audit started ################
                                                    2010-08-14T15:34:12+01
/vob/test/nmvfs/makefoo <2010-08-14T15:26:10+01>
$ touch 08141530 /tmp/makefoo
$ clearmake
 /vob/test/nmvfs/makefoo
```

ClearCase takes responsibility for auditing the identity of the tool, using the time stamp for a view private object (and the version for a checked in element).

We are anyway left now with only one alternative to eliminate: hardcoding the version in the tool's pathname. We must reckon that, in the above scenario, this would have worked just fine.

Referential transparency

The situation just described gets worse if one considers having to switch back and forth between several versions of the tools.

Without ClearCase, the traditional solution is to rely upon naming, that is, to hardcode the version of the tools into the paths used to access them.

Of course, one needs then to produce one's derived objects under names or paths matching this hardcoding.

Such a solution may unfortunately be considered in simple cases, before the combinatorial explosion of tool variants gets acknowledged. At that time, one has already invested in generic complexity of the build system (using macros, and/or generating the build scripts).

ClearCase offers a far more elegant solution by decoupling the versioning dimension from the common namespace. Allowing the versions of tools to be selected via the config spec makes it possible for static files (build and other scripts, documentation) to be reused among different software configurations, thus reducing dramatically the global number of artifacts.

Referential transparency—the property for the same name to map to different instances of a resource in different contexts—decreases the artificial complexity and helps making essential differences emerge.

Switching versions of tools, that is, switching software configurations, becomes an incremental task, concerning only a few files. The threshold for experimenting or tuning the environment gets lowered, and the overall quality improves.

With naming-based solutions, the smallest change is an all-or-nothing challenge. This results in postponing fixes, and bundling changes together, which in turn produces massive changes and long term instability. It results in matching investments to expectations, instead of making choices based on real experiments.

Flexibility

It is quite typical that different components of a large system, different applications, do not evolve synchronously with respect to their use of tools. One application may meet a limitation or a bug, which makes it critical for it to move onto the next release of the tool. Feeling the pain, the team is motivated to invest the effort to validate it and to adapt their own software to the API changes. Other applications will, on the contrary, be fighting with different problems and satisfied with the functions offered by the current version.

Adopting a new version, or a new tool in replacement for an existing one, is not a trivial enterprise, and typically takes time and requires making changes. It is highly beneficial that this may happen in-place, without requiring one to clone the existing development environment. Once the evaluation has happened, assuming the decision was made to move onto the new tool, one ought to be able to use the results of the effort without having to pay the fee a second time.

These remarks fight a common bad practice to consider the SCM repository as a certified archive, in other words, as a place to store only *official* artifacts, ones that have gone through some kind of formal approval procedure. As we already stated, management is most useful before the artifacts have been validated. Vobs are the natural place to store the candidates of such a validation, and the validation, as any kind of release, should happen *in-place*.

Tool fixes

The boundary between tuning a tool version and making a new one is itself neither clear-cut nor stable: tools typically have to be configured, and one may even have to maintain workarounds for defects, before a vendor may provide fixes.

Tuning typically depends on usage patterns, and therefore happens naturally late in the development, and in small increments, with trial and error. Clearly, this should defeat a tool maintenance strategy based on naming: with such a strategy, in order to maintain the ability to rollback one's changes or to compare the results with previous ones, every change should result in duplicating a large portion of the tool hierarchy as well as the set of products. This may still be considered in simple cases, but is utterly non-scalable in the presence of combinations of tools and/or platforms.

Indirect dependencies

Third-party tools are themselves built from third (or is it fourth?) party tools. It is commonplace to find the same dependencies crop via different paths, and this is the cause of many consistency concerns: upgrading one tool may force to upgrade several others in order to build up a coherent virtual baseline for their combined dependencies. This is a non-trivial task, in which few vendors will assist, and for which none of them will take responsibility. In fact, this is an excellent argument in favor of the Open Source model.

Managing the tools under ClearCase certainly makes it easier to build up such baselines, and evolve them. You may help ClearCase to help you, by sharing the tools that would be embedded, instead of dumping in various places many instances of the same. This will help detecting problems, which is often a prerequisite for fixing them.

Now, the notion of dependency used in this context need not (and often cannot) be as strict as the SCM notion of dependency (based on identity). There comes the weaker concept of *compatibility* (especially of *backwards* compatibility). One will often have to trade for a common set of tools, adjusted from a compromise between otherwise incompatible tool requirements.

MultiSite replication

Tools maintained in vobs may obviously be replicated, and thus shared between different sites. This leverages the installation work, which may be used to distribute the effort, or... to distribute the pain. It is advisable to opt for a collaborative model, and let sites feeling first the need for a new tool or an upgrade do the investment of installing and validating the new item, rather than for a centralized model, in which such upgrades have to be ordered. The obvious danger of not doing so is that it will not prevent the other sites from installing and using the new release: they will only go underground, and it will be harder to converge back. The more you control, the less you manage!

Our recommendation is here again to remember to depend in config specs on *local* labels, that is, to use remote ones (applied at another site) only as the basis for moving one's own. The rationale is the intrinsic asynchrony of replication, leading to the fact that there is never any guarantee of consistency in the local image of a remote baseline: at every synchronization not only the present, but the *past* may change (we looked at this in *Chapter 5*, *MultiSite*, under the section named *Labels*).

How?

Installing a tool in a vob takes place in two phases:

- The installation proper, to a local ClearCase client
- The import from there into the vob

The import phase can often use `clearfsimport`, from the ClearCase distribution, but in many cases goes much easier with ***synctree***.

The installation phase depends mostly on the way the tool is packaged and distributed. It is however often necessary to consider the import phase first, because it may influence the parameters used to configure the installation (the paths for a start).

We'll start with an example of a tool, the one we announced already in *Chapter* 1, *Using the Command Line*: perl. We'll try to generalize from it.

Example—perl installation under a ClearCase vob, with multi-platform support

Get a perl distribution for the first platform, for example, Linux. But here comes the first choice: binary or source distribution?

Perl will embed the paths to its modules in an `@INC` variable. It will also record the compiler used to build it, so as to use the same for building binaries that might be part of modules to be installed on it (if they so require).

These aspects guide us to build perl ourselves, from a source distribution. However, in the particular case of Windows, it is possible to use a binary distribution and to configure it to use a vob path (associated to a drive letter).

Building perl itself is straightforward, if your environment meets the requirements that are clearly defined in per platform readme files. You may of course want to use the compiler and other tools from a vob, and thus to import them first. You may get into chicken and egg problems, and to solve these, you would have to use binary distributions first.

Before we install it, let's anticipate about what the maintenance will be. Let's thus start by the end, by importing or updating individual modules on top of this vob installation of perl.

Importing CPAN modules

Download the Perl module from CPAN:

```
$ wget http://search.cpan.org/CPAN/authors/id/D/DS/DSB/###################
                                                    ClearCase-Argv-1.49.tar.gz
$ tar xzf ClearCase-Argv-1.49.tar.gz
$ cd ClearCase-Argv-1.49
```

First install locally, but using perl from the vob (which has been imported there already, see below):

```
$ which perl
/vob/tools/perl/bin/perl
$ perl Makefile.PL PREFIX=/home/marc/tmp
$ make
cp Argv.pm blib/lib/ClearCase/Argv.pm
Manifying blib/man3/ClearCase::Argv.3
$ make install
Installing /home/marc/tmp/lib/site_perl/5.10.1/ClearCase/Argv.pm
Installing /home/marc/tmp/man/man3/ClearCase::Argv.3
Writing /home/marc/tmp/lib/site_perl/5.10.1/i686-linux/auto/##############
                                                  ClearCase/Argv/.packlist
```

```
Appending installation info to /home/marc/tmp/lib/perl5/5.10.1/ #########
                                                    i686-linux/perllocal.pod
```

Import from there:

```
$ ct setview v1
$ ct setcs cs/linux
$ ct catcs
element * CHECKEDOUT
element * .../imp/LATEST
mkbranch imp
element * /main/LATEST

$ sb=/home/marc/tmp; db=/vob/tools/perl
$ synctree -sb $sb -db $db -/ipc=1 -reuse -vreuse -ci -yes -label LINUX \
 lib/site_perl/5.10.1/ClearCase/Argv.pm man/man3/ClearCase::Argv.3
```

This method is easy, apart from the issue of updating the `perllocal.pod` and the `.packlist` files.

New information is appended to `perllocal.pod` for every module installation. The problem is that the paths recorded there must be manually fixed after importing to the vob. The file is found in the platform-dependent tree (which is a reason to share this tree between the platforms).

```
$ perl -pi -e 's#/home/marc/tmp#/vob/tools/perl#' \
 /home/marc/tmp/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ ct co -nc $db/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ cat $sb/lib/perl5/5.10.1/i686-linux/perllocal.pod \
 >>$db/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ ct ci -nc $db/lib/perl5/5.10.1/i686-linux/perllocal.pod
$ ct mklabel LINUX $db/lib/perl5/5.10.1/i686-linux/perllocal.pod
```

The `.packlist` files contain the list of files belonging to a package. They are therefore more stable than `perllocal.pod`. They are also found in platform-specific trees.

```
$ cat \
 ~/tmp/lib/site_perl/5.10.1/i686-linux/auto/ClearCase/Argv/.packlist
/home/marc/tmp/lib/site_perl/5.10.1/ClearCase/Argv.pm
/home/marc/tmp/man/man3/ClearCase::Argv.3

$ perl -pi -e "s#/home/marc/tmp#/vob/tools/perl#" \
 ~/tmp/lib/site_perl/5.10.1/i686-linux/auto/ClearCase/Argv/.packlist

$ synctree -sb $sb -db $db -/ipc=1 -reuse -vreuse -ci -yes -label LINUX \
 lib/site_perl/5.10.1/i686-linux/auto/ClearCase/Argv/.packlist
```

Even if it is the easiest method, one often has to import different parts separately: scripts, perl modules, auto-split directories, man pages... For some of them, one may let the clean up to the `-rm` option of synctree (to remove some of the elements if they are not present in the current version of the module), for others it's impossible: it depends whether the destination directories are shared with other modules or not.

The vast majority of CPAN modules are perl only, hence, platform independent. They can thus be imported once for all the platforms. We must only avoid introducing accidental factors of dispersion. Two come to mind at this stage:

- Avoid platform-specific branches
- Avoid platform-specific directory names

For the platform-dependent modules, one could use:

- Separate branches, say, `imp1`
- Platform specific labels—`SOLARIS`

We have to admit that we didn't configure perl to avoid the platform-specific names for a few directories. We could have done it by first disabling the platform-specific lib paths in the perl installation and replacing them with a common path `arch` such as `lib/site_perl/5.10.1/arch` (by using the `-Darchname=arch` option for `Configure` script, see below). And then we could even specify such common paths as `INSTALLSITELIB` and `INSTALLPRIVLIB` parameters when building our module, in an attempt to make our imports easier and thus less error-prone:

```
$ perl Makefile.PL PREFIX=~/tmp \
 INSTALLSITELIB=~/tmp/lib/site_perl/5.10.1/arch \
 INSTALLPRIVLIB=~/tmp/lib/5.10.1/arch
```

Instead, we did share the platform-specific directory names, by renaming them as configured, so that in our experience, the `perllocal.pod` and `.packlist` files mentioned above are shared at the element level but unfortunately not (at least for `perllocal.pod`) at the version level:

```
$ ct setview v2
$ ct setcs cs/solaris
$ ct catcs
element * CHECKEDOUT
element * .../imp1/LATEST
mkbranch imp1
element * LINUX
element * /main/LATEST
```

```
$ ct co -nc /vob/tools/perl/lib/site_perl/5.10.1
$ ct mv /vob/tools/perl/lib/site_perl/5.10.1/i686-linux \
 /vob/tools/perl/lib/site_perl/5.10.1/sun4-solaris-thread-multi
$ ct ci -nc /vob/tools/perl/lib/site_perl/5.10.1
$ ct mklabel -rep SOLARIS /vob/tools/perl/lib/site_perl/5.10.1
```

We also renamed the `/vob/tools/perl/lib/perl5/5.10.1/i686-linux` in the same way and then we proceed with importing the Solaris version of the module to the vob, in the same way as described earlier for Linux—in `imp1` branch and we label it with `SOLARIS` label. But this discussion anticipates on the *Upgrading the distribution* section that will come later in the chapter.

Installing the Perl distribution

Get the Perl distribution (the source code) for Linux (version 5.8.8 is chosen on purpose: it anticipates upon the section on upgrading):

```
$ wget http://www.cpan.org/src/perl-5.8.8.tar.gz
$ tar xzf perl-5.8.8.tar.gz
$ ct setview v1
$ ct startview v2
$ ct mkbrtype -nc imp@/vob/tools
$ ct setcs -tag v1 cs/linux
$ ct setcs -tag v2 cs/linux
$ ct catcs -tag v2
element * CHECKEDOUT
element * .../imp/LATEST
mkbranch imp
element * /main/LATEST
$ ./Configure -des -d -Dprefix=/vob/tools/perl
$ make install
$ synctree -sb /vob/tools/perl -db /view/v2/vob/tools/perl \
 -/ipc=1 -reuse -vreuse -ci -yes -label LINUX
```

After the successful import, remove the `v1` view-private directory `/vob/tools/perl` and its content, which is currently eclipsing the imported versions:

```
$ ct des /vob/tools/perl
View private directory "/vob/tools/perl"
$ rm -rf /vob/tools/perl
$ ct des -fmt "%n %l\n" /vob/tools/perl
/vob/tools/perl@@/main/t/1 (LINUX)
```

Repeat for other platforms, for example, Solaris, Windows. At import time, one needs to modify the config spec to take the previous baseline, identified with the `LINUX` labels (as we have just showed above in the *Importing CPAN modules* section).

For those, you'll need to use different label types (for example, `SOLARIS`, `WINDOWS`), and to add one option (already mentioned for modules) to the `synctree` command: `-rm`. This is to remove files (obviously, it was redundant in the very first import) that would not exist in the new source base.

An additional flag is often useful: `-lbmods`, to apply labels only to the versions modified. We choose here to apply one full label per platform though, and thus not to use this flag.

At least with the `Configure` line presented above (the one we used ourselves), we obtain some sub-trees as follows:

```
lib/site_perl/5.8.8/sun4-solaris-thread-multi
lib/site_perl/5.8.8/i686-linux
lib/5.8.8/sun4-solaris-thread-multi
lib/5.8.8/i686-linux
```

One option would be to create hard links for lib directories:

```
$ cd lib/site_perl/5.8.8
$ ls
i686-linux
$ ct co -nc .
$ ct ln i686-linux sun4-solaris-thread-multi
$ ct ci -nc .
```

We suggest rather renaming the platform-specific directories for every platform. It makes it easier to compare the different installations, and even to share some files there (see earlier section named *Importing CPAN modules*). To do so, one needs to prepare the *destination base*, that is, the vob copy, prior to running *synctree*.

Some `Configure` options to consider (see earlier):

- `-Dinc_version_list=none`: This disables an option offered by perl to keep historical versions of some scripts in release-specific directory trees
- `-Darchname=arch`: This standardizes the directory names above (but `thread-multi` is still appended), that is, the platform-specific paths, `sun4-solaris-thread-multi` and `i686-linux`, are replaced by the common one, `arch`:

  ```
  $ ./Configure -des -d -Dprefix=/vob/tools/perl -Darchname=arch
  $ make install
  $ ls /vob/tools/perl/lib/site_perl/5.8.8
  arch
  $ ls lib/site_perl/5.8.8
  arch
  ```

Upgrading the distribution

The situation is slightly different when it comes to upgrading this installation, say from 5.8.8 (which is already under ClearCase) to 5.10.1.

We suggest that prior to upgrading the first platform (for example, Linux), the Perl version-specific directories are renamed manually just as we did for the platform-specific ones:

```
$ ct co -nc /vob/tools/perl/lib/site_perl
$ ct mv /vob/tools/perl/lib/site_perl/5.8.8 \
 /vob/tools/perl/lib/site_perl/5.10.1
$ ct ci -nc /vob/tools/perl/lib/site_perl
$ ct des -fmt "%n %c" lbtype:LINUX1
LINUX1 Perl 5.10.1 for Linux
$ ct mklabel LINUX1 /vob/tools/perl/lib/site_perl/5.10.1
```

Then one can proceed with installation of the new (5.10.1) version and importing it to the vob. Note that one needs to be careful using the `-rm` flag: it is appropriate for the distribution part, but not under the `site_perl` directories, where it would result in removing any CPAN modules that have been installed in the meanwhile.

At the point of importing the second platform, one faces however a dilemma: what baseline should one use?

- The previous baseline for the same platform (which was for 5.8.8)?
- The baseline for the new version, but for the platform just imported?

This is where the `-reuse` and `-vreuse` options of `synctree` make the difference. There is more on this in the section on *Branching and labeling*.

Installation

Time now to try to generalize... or as we'll soon notice, to fail to do so, facing the specificity of each tool. We can only draw our reader's attention to a few issues that need to be considered.

The perl model: extract tar, `Configure` (or `configure`, for example, for GNU tools), and `make install`, is of course not the only one.

The choice of installing binaries or sources may be met with a packaging tool other than tar: for example, with rpm. In the case of Linux, one typically installs first a binary distribution, and it is only when wanting to debug and fix a problem, and possibly to contribute the fix back, that one installs a source distribution, and builds from there. In any case, there are strong reasons *not* to attempt to build the tool under clearmake: it is unlikely that the build system of the tool will meet the requirements that would make builds avoidable under winkin and thus manageable via derived objects. Changes one would make to it could not be contributed back, and would thus constitute a *fork*. Not sharing the results as derived objects, one would have to *stage* (that is, fall back to the model of the binary distribution).

In short, one has to have a good reason to manage the *sources* of the third-party tool under ClearCase and we do not see fit to make it a rule.

The installation typically uses a packaging tool specific to the package format, typically platform dependent (`pkgadd` on Solaris, `rpm` on Linux, `swinstall` on HP-UX, and so on), and the import is done from the installation tree. There is typically an option (such as `-R` for `pkgadd`) to relocate this tree to a path under a vob, but whether this is possible or not depends also on the product itself: some products are hardcoded to be used from certain paths. It may still be possible to import them in a vob, and to use them via symbolic links (which will only work under a view context).

The install tool may require *root* privileges, and the product may depend on certain access rights.

Note that a package targeted to multiple platforms cannot be assumed to install identically on every one of them, so that one could install it and import it only once.

Let's take the example of a Solaris package, intended for both *sparc* and *i386* platforms. The `pkgmap` file found when extracting it drives the installation with `pkgadd`. It contains lines that describe file objects to be installed, one per line. It is however possible and customary to use macros in the paths that are set in the `checkinstall` script, for instance based on the value returned by `uname -p` (precisely, `sparc` or `i386`). In the simplest case, this mechanism will drive the creation of a symbolic link pointing to binaries of the suitable architecture.

Products may themselves contain symbolic links (internal or external), which may have to be adjusted to work under the vob. Again, synctree has an option to assist with this: `-rellinks`.

Using a tool from the vob, one must remember to adjust some environment variables used by various utilities (`MANPATH` to access manual pages, `LD_LIBRARY_PATH` or platform equivalent to access shared libraries, `JAVAHOME`, and so on).

As in the perl case, the first installation is usually simpler than any subsequent ones. It is however the place to make wise decisions, concerning the paths. Fortunately, ClearCase will make it possible to reconsider those decisions, to learn from one's mistakes, and to tune the directory layout to favor **stability**, even if one failed initially to do so.

Subsequent installations of a product may take the existing configuration into consideration. Patch installations will even require an existing installation. Typically, the vob image of the product is not directly suitable: it cannot be overwritten, even with root privileges. One needs therefore to be able to produce a writable copy of the vob image, suitable for the installer, and using the same configuration as for the original install. One option is to keep the original copy after importing to a vob from it. Another, perhaps a better option, is to use a dynamic view with `-none` rules in its config spec, and a view private copy of the vob contents. For copying the vob contents to the view, in case `cp -R` is not suitable on the current platform, one may use the old idiom:

```
$ tar cf - . | ( cd /view/copy$(pwd) && tar xf - )
```

Import

Now that we do have a local installation (after either doing it for the first time or applying a patch, or even in some cases just extracting an archive), our next goal is to import it as such into the vob.

Minor checks prior to importing

We already mentioned, in the earlier section, about the possible issue of symlinks, which may have to be adjusted. Apart from those, special protections may have to require attention. First, the groups used must be declared to the vob. The most exotic protection schemes (such as those used in the vob storages for the contents of the `.identity` directory) may not be supported. Neither can devices, sockets, pipes, and such objects.

It is a good idea to check (and fix) the access rights in the local installation (protections may be corrected in the vob, but the resulting events may be filtered from the synchronization to other replicas, depending on permissions preserving settings): executable files should have the executable bits set (this might not be the case if the image was extracted from a Windows archive); directories should be executable as well, and usually group-writable; file objects should be world readable. Contradictions between these rules and "security" concerns should be handled case by case, examining possible consequences.

Branching and labeling

Traditionally, separate branches types would be recommended:

- For releasing tools
- Local configuration and changes
- Binaries built out of sources

This is not necessary if using synctree with the `-vreuse` and `-label` options. In this case, the goal of selecting versions exclusively with labels, and being agnostic to the actual location in the version tree, may be restored.

The script will reuse existing versions, even if they are not selected: if it finds a suitable one in the version tree, it will apply the new label there instead of importing a duplicate.

Each tool release can be labeled by a fixed label (which may be partial), for example `GCC_2`.

A distinct common *floating* label, for example `TOOLS`, can be used for publishing the whole tools set: it will be applied over a chosen fixed label for each selected tool.

Issues during the import

One issue not automatically handled by the import tools (neither clearfsimport nor synctree) is that of incompatible requirements on the type of some elements between the version already in the vob and the one being imported. This could typically be met with HTML pages (the old version had short lines and could be imported with an old, text based, element type, whereas the new one has lines above 1024 characters and requires a new "binary" element type), but also, for example, with GNU info files.

These problems are difficult to predict and will typically be detected only as import errors, reported by the tool but always easy to miss from verbose transcripts. The elements will be left checked out. The fix requires manual intervention, usually with `chtype` and the appropriate new type. Synctree will attempt to cope with transitions between symlinks and files or directories. Cases in which the same name is assigned between two releases to a file and a directory will require manual removal (`rmname`) of the offending object.

Operating system

ClearCase is not available before late runlevel 2 (3 on HP-UX). This means that everything needed before that cannot be maintained (exclusively) in vobs. Of course, it is possible to switch, once ClearCase is available, and for resources it doesn't use itself, to copies from the vobs (some boot sequences use a distinct local file system in single user mode, not available later) via symbolic links. A view context is of course required to access them. One needs a pair of scripts used at boot and shutdown time to set and reset those links, renaming away and back the local versions of the resources.

This makes maintenance tasks heavier (for example, installing patches) and more disruptive. The set of resources used from the vobs is also not guaranteed to be stable.

Versioning under ClearCase also makes the whole system dependent on license availability. Even users who would only *use* the tools, for any other need than software development, would need a ClearCase license once the tools are stored in a vob. Additional cost may understandably become a concern.

There are other limitations to what may be versioned. For instance, some resources are inherently unique, thus can only be found in one version, for example, sockets, but also license and other databases. One may also think of hashes of tools, for commands such as `locate` or `man`.

Finally, there may be a performance penalty, an impact on storage requirements, on access control, on maintenance tasks, and so on. All these possible issues must be studied and compared with the benefits described before. The best tradeoff will heavily depend on the environment.

Shared libraries

Shared libraries are a special case: some of them will be used by tools needed to set up the environment for running ClearCase. In fact, cleartool itself uses `libC.so` (Linux/Solaris extension). Since they are shared, it is enough that one of their clients needs to be available outside of a ClearCase context, so that the shared libraries must be used from standard storage.

An interesting aspect of shared libraries is their versioning. The dynamic loader implements a crude version control system, based on names recorded at link time in client applications. In order to allow applications to link with the latest (that is current) version of the shared libraries, without relying on the developers having to be aware of it, a stable name is maintained as a symbolic link to the version-decorated name of the actual file:

```
libfoo.so -> libfoo.so.3
```

As it happens, it is often older versions of shared libraries that are in use in operating system utilities: older at least than the versions that are offered for linking to the developers. This means that one may often maintain the recent versions under ClearCase, and leave older versions in the system directories. Or rather, that there is no conflict in installing the latest patches in the vobs, without upgrading the hosts themselves.

One digression while dealing with shared library versions is that whether in the standard scheme it is the symbolic link which bears the stable name, and the actual file which bears the version decorated one, this is actually not required either at link time or at run time. It may thus be more convenient in the vobs to maintain the name pairs in the other way around, at least when developing shared libraries: producing new versions of the shared libraries under a stable name, and creating, as needed version decorated symbolic links to them. The name recorded in client applications must only be the version decorated one:

```
$ ld bar.o -G -o libbar.so -h libbar.so.1
$ ln -s libbar.so libbar.so.1
$ gcc -o foo foo.o -L. -lbar
$ export LD_LIBRARY_PATH=/vob/test/libso
$ ldd foo1 | grep libbar
 libbar.so.1 => /vob/test/libso/libbar.so.1
$ ls -l libbar.so.1
lrwxrwxrwx 1 marc jgroup 10 Aug 21 19:10 libbar.so.1 -> libbar.so
```

It may be worth stopping for a while on this issue of shared libraries, and considering it in a wider perspective: what this means is that the run-time environment is already able (in a limited way) to cope with multiple versions of the same resources, that is, it doesn't completely fulfill the basic assumption allowing one to consider it as one (consistent) software configuration: the exclusion principle, *at most one version of any resource*. Or in other terms, there is not one monolithic and consistent software configuration, but an overlapping collection thereof. Let's note that this is only a tendency taken to a much wider scope with for example, the Eclipse repository. We shall mention this again in *Chapter 12, Challenges*.

A different strategy to maintain shared libraries is to deliberately break the overlapping between the development and the run-time environment, thus abandoning the idea of *sharing* them: maintain a copy of the shared libraries in the vob, instructing the linker to write the standard path (instead of the one actually used at link time, pointing within the vob) into the executables. This means that the result of builds may not be usable in the run-time environment, which one may consider as an extension of a concept of cross-compilation.

Licenses

Some tools have host-based licenses, so that sharing them in vobs will not work. It is sometimes possible to store the license keys for every host in a standard path, and to create a symbolic link to it from the vob.

The tool will be usable only on host for which the link to the license key will not be dangling.

MultiSite and binary elements

The previous discussion about licenses applies to more license schemes (not only host-locked) in presence of MultiSite, but the treatment remains similar.

MultiSite brings some other concerns, which would be common with staging (if practiced), and relate to storing and shipping of large binaries.

The mere size of binaries, together with their "opacity", fight continuity: every version takes a large storage, events concerning them are large and cannot be split (or only under certain conditions), resulting in large sync packets that take a long time to ship, require large buffers, and result in frequent errors.

The only common option is compression, but this is a two-edged sword, as time is needed to compress and decompress and it often requires yet larger buffers (and results in duplication at the container level (refer to *Chapter 9*, *Secondary Metadata*, about the storage of binary types).

In short, ClearCase can version binaries, but doesn't do miracles: tradeoffs come near. In particular, there is often very little value in storing packages or zipped archives in vobs.

Labels, config specs, and multiple platforms

We already mentioned using (site) local labels in the context of MultiSite, and understand now that handling binaries makes it even more necessary for tools than for source code. Let's add that (remote) labels crossing vob boundaries (either *global*, or just sharing the same name and thus the same rules in config specs) are even more in danger of facing transient inconsistencies.

We also saw that tools are often to be dealt in large baselines of consistent (or at least compatible) releases. It is often convenient to name these baselines with compound label types, with the effect that these types suffer from combined volatility: they change as soon as any tool changes, which is sometimes not predictable (it depends on defects and fixes by other organisations).

This is a typical case for *floating* labels (for developers' convenience, and minimizing errors). It is however important to retain underneath fixed labels for the individual tools.

Both the floating and the fixed labels should be maintained for every platform. As seen previously (*Referential Transparency*) one ought to share as much as possible between the various platforms, under the same paths, and avoid duplicates and *evil twins*. Tools are at the bottom of the development chain, and thus in a strategic position: errors made at that level will easily multiply and result in combinatorial explosions, missed opportunities for sharing results, and duplicated work.

Special cases: Java 1.4.2_05 on Linux

Exceptionally, it is possible that some tools cannot be used under ClearCase.

We know only the case of Java 1.4.2_05 and above versions on Linux. In November 2004, Sun made a change to the Java launcher, exclusively on the Linux platform (although the same thing could have been made, for example, on Solaris), and which broke among other things (for example the use of `LD_LIBRARY_PATH`), launching java from a vob. The change required that the `lib` directory containing the shared libraries used by the binaries would be co-located to the directory tree returned from the value of `/proc/self/exe`. This was an arbitrary assumption, defeated by the fact that in the context of ClearCase, the path returned would point inside the cleartext pool.

Working around this was allegedly possible by implementing an interposer library to fake a failure of the `readlink` system call, thus falling back to another mechanism. This could affect other tools and contexts.

In practice, the change made it impossible to use Java from a vob on Linux.

This situation prevailed for at least a few years. We are unclear about the ***current status***. Maybe the IBM JRE, bundled with every Rational product, could offer a solution.

Naming issues: acquisitions, splits, mergers

Our last remark will be to warn against hardcoding in the installation paths names beyond one's control, and which are not strictly necessary.

Mentions to companies in particular are now-a-days particularly volatile.

Let's remember, for example, that ClearCase was not always a product of IBM: one might have had to rename occurrences of *Rational*, after *Pure*, after *Atria*.

Summary

We hope the long list of concerns we reviewed in our second part did not overshadow the bright perspectives opened in the first one!

Tool maintenance is a good example of the *no free lunch* reality prevailing in software development. The issues have to be addressed, and the sooner they are, the better: these are investments that pay back! Once the base environment is in place, upgrades will soon feel like routine. The comfort of being able to switch back and forth with only config spec changes between different releases of a tool will feel invaluable: was a bug fixed? did the performance change? Is this new tool compatible with the rest of the portfolio or where to trace the next baseline? These are questions that are much better answered, and with incomparably more confidence, by actual testing than by reviewing release notes!

In the next three chapters, we shall continue to fiddle with administration issues, never losing the user's perspective: what's behind the scenes, and what should there be, so that my needs are fulfilled in the best possible way? What can I, and what should I, do to make it work even better?