Using the Shape Toolkit for Cooperative Software Development
– A Tutorial –

Axel Mahler
Technische Universität Berlin
Axel.Mahler@cs.tu-berlin.de

July 1993

shapeTools_TUT-1.3

INTRODUCTION

The ‘‘Shape Toolkit’’ (we just call it *ShapeTools*) is a collection of tools that help to establish effective control over the intricacies of the *change process* in software development projects. The keywords here are *version-control* (VC) and *software configuration management* (SCM). As an additional keyword (actually a current buzzword) we shall throw in *computer supported cooperative work* (CSCW). ShapeTools will be of most value to medium software development projects. While one-man/woman projects might benefit from using the toolkit, its most significant effect is coordination (isolation and controlled consolidation) of change activities within multi programmer projects. A shape-supported development environment offers effective control over the change process: programmers can freely work on components without interfering with other programmers who need access to stable versions of these components. Shape and the version control system give plenty of flexibility for implementing policies regarding issues such as when changes shall become visible, to whom, or when they become effective.

SCM is a rather complex and elaborate discipline for managing the development and maintenance of software systems throughout their entire life cycle. Just installing ShapeTools cannot substitute a thorough understanding of how SCM works. However, we hope that ShapeTools will help to approach SCM by carefully automating various of the tedious everyday tasks in software development management. This tutorial is mainly intended to describe the ShapeTools system at hand. It cannot replace a textbook on SCM. We will do our best, though, to emphasize the way, how ShapeTools can be applied to enhance the capability of teamwork development projects. In order to do so, we will present a cocktail of functionality oriented feature explanation, and exemplary application scenarios. Although the examples will fall short of describing comprehensive SCM environments, they will go far beyond everything that can be done with crafty combinations of Make and SCCS or RCS.

1. The ShapeTools Toolkit Concept

ShapeTools is a *toolkit*, not a monolithic, ready-made, total system. The main parts are a couple of programs that can be called from the shell command line, and a library of C-functions that provide a fine-grain programmatic interface to the *object base* of ShapeTools, the *Attributed File System* (AtFS)*. The ShapeTools commands are carefully designed to fit the Unix standard-input/-output concept and thus can be easily combined with all other standard Unix tools. The numerous functions are designed to provide an intermediate level of functional granularity in order to allow easy and rapid construction of specific SCM functionality, custom made for your project. Although all commands can be used directly from the command level without any problem, it generally makes most sense to invoke them from shell-scripts or via the shape program. The ultimate idea is to use an integrated development environment or a user interface system (e.g. GNU-Emacs, or an X-Windows based UIMS) to activate specifically designed, more complex SCM functions that are implemented with ShapeTools from menus or even implicitly.

If you work as a toolsmith for a development project, we want to encourage you, to design your own software development environment based on the ShapeTools toolkit. As mentioned above, this will require careful analysis of the actual cooperative work patterns in the project. Once the basic teamwork processes are understood, it shall be easy to figure out how these can be effectively supported. We believe that you will find that ShapeTools makes it easy to develop a teamwork support environment. When setting up such a teamwork environment, the toolkit nature of the ShapeTools system implies some additional work because the individual functions are somewhat rudimentary. However, when designing ShapeTools, we deliberately took this approach because software configuration management is typically very project specific. Providing more complex, specific functionality might make life easier for some project support engineers, but at the same time the system might become almost useless for many other projects. There is quite some expert work required to build suitably customized SCM environments, and you, the reader, are probably just about to be that expert for your organization.

Throughout this tutorial we will examine a number of typical elementary development processes and show how these can be effectively supported by ShapeTools. These processes are:

– the reserve-modify-build-test-deposit cycle
– the individual-development process (development/unit-test)
– the incremental system test process (integration-test)
– the management of parallel system variants
– change request driven development/maintenance
– automated system release.

The remainder of this tutorial is split into two parts: the description of the ShapeTools version control system, and the description of *shape*, the toolkits main configuration management tool. Both parts can be read independently from each other, but we recommend to start with the version control system in order to get acquainted with the concepts and capabilities of the AtFS object base. The tutorial will be even more instructive, if you have a copy of the ShapeTools man-pages handy.

---

*) AtFS is a user-level extension of the Unix file system. There are no modifications at all to the file system code in the operating system kernel.

PART I — SHAPETOOLS VERSION CONTROL

The version control commands of the shape toolkit are useful to manage multiple revisions of source objects (usually ASCII text files but not limited to) as well as multiple instances of derived objects (usually produced by compiling one or more source objects). The version control system allows storage, retrieval, logging, and identification of revisions. All objects are stored in an object base that resides in a subdirectory with the name AtFS.

The ‘‘thingies’’ we are dealing with are called *objects* (in full length: *Attributed Software Objects*, ASOs) because they are more than just files. They have a memory for states they have been in in the past, versions. And even more than that, objects can be annotated with *attributes* supposedly describing various aspects of an object or object version. However, it is convenient and straightforward to think of these objects as files (keep in mind, though, that *versions* are immutable, while files can be altered). Throughout this document, we will frequently use the term *source object* synonymous with ASO.

Consequent use of version control is the prerequisite for configuration management and software maintenance. Using the version control system also eases cooperation within a programmer team, and even supports individual work styles, such as saving intermediate work results that may be backed up to, in case that particular modifications turn out to be unsatisfying.

The basic user interface consists of a number of simple commands that are intended for building more sophisticated project support systems on top of them. However, simple use of the version control system is straightforward. The novice user only needs to know three of them: *save*, *retrv*, and *vl*. **save** stores the contents of a specified file as a version in the object base. **retrv**, short for ‘‘retrieve’’, retrieves the contents of versions from the object base. **vl**, short for ‘‘version list’’, lists the objects in the current object base and the files – typically in the current directory. The current object base is the *AtFS* subdirectory of the current working directory. While these three commands may suffice to serve the needs of single-programmer projects, ShapeTools’ version control system also offers more sophisticated support for larger projects that employ programmer teams.

2. Functional Overview of the Version Control System

Storage and retrieval of multiple revisions of source objects. All revisions are stored in a space efficient manner. Changes no longer destroy possibly worthwhile status of the original, because previous revisions remain accessible. Revisions can be retrieved from the object base according to revision numbers, symbolic names, states, or arbitrary attribute patterns (save, retrv, vcat, vl, vfind, vbind).

Storage of arbitrary module- or version related information along with the objects. The object management system of ShapeTools allows to attach any number and kind of *attributes* to particular objects and object versions. This is very useful for many purposes, such as passing information to other tools or making certain expensively computed properties of an object explicit and persistent.

Built in automatic status model for revisions. Each revision is in one of the states busy, saved, proposed, published, accessed, or frozen, according to its supposed quality and access status. State transitions happen (semi-) automatically upon invocation of certain commands (save, sbmt, publ, accs, frze, vadm promote, vadm unpromote). While this status model offers some basic help for organizing larger, multi-programmer projects, it does not check authorizations that should be required in order to perform certain status changes. This has to be realized by other functions that can be implemented using the basic version control commands (e.g. by scripts or environment support).

Maintenance of a complete history of changes. Besides the contents of each revision, the version control system stores the author, date and time of save operations, as well as a log message summarizing the changes as attributes of the newly created version object. The version control system distinguishes saving of intermediate work (save) and submitting final work results to the project (sbmt). There are also hooks provided for approvals by quality assurance functions, such as distinguishing proposed and published version status. Upon submission of a new version, the log message is initialized with the description of the planned change which is acquired upon reservation of the update privilege for the design object (save, vlog, vdiff, vadm).

Resolution of access conflicts. When two or more programmers wish to modify the same source object, the version control system alerts the programmers and tries to keep them from mutually corrupting each others modifications (retrv, vadm).

Comprehensive support for configuration control. Revisions can be assigned symbolic names that typically are shared among all component revisions of a configuration. Symbolic names can be used as mnemonic *aliases* for version numbers, and may appear anywhere a version number is allowed. Moreover, the version control system is fully integrated with shape, the configuration identification and build tool (save, vadm, vl, vfind).

Automatic identification of revisions with name, version number, save date, author etc. The identification is like a stamp that can be embedded at an appropriate place within the contents of an object. This is useful for marking derived objects (object files) with a reference to the respective source-object versions thus making it possible to identify the configuration of a given executable program (retrv, afsit, ident, shape).

3. Getting Started

Before you can use any of the version control facilities, you should create a version repository which resides in subdirectory ‘‘AtFS’’. The mkatfs command does this for you (things that you type are printed in boldface).

```
$ mkatfs
AtFS repository initialized.
```

Now, suppose you have a file ‘‘xyzzy.c’’ that you wish to place under version control. To do this, just invoke the save command.

```
$ save xyzzy.c
xyzzy.c:
creating archive
How about describing the purpose of this document ? [yes]
<starting favorite editor>
xyzzy.c[1.0] saved.
xyzzy.c removed.
```

This command creates an entry for ‘‘xyzzy.c’’ in the repository. Before the first version of an object is created, save asks for a short description of the objects purpose. This description should be a synopsis of the contents. All later save commands will ask for a log, which should describe the changes that have been made. If you issue the command vl, you should get

```
$ vl
xyzzy.c[1.0]
```

indicating, that you now have a version, revision 1.0 of ‘‘xyzzy.c’’. After saving a revision of a file, save removes the busy version in order to make the object available for possible update by somebody else. There is of course a command switch that retains the busy version, and sets a lock on the object history. A lock represents the exclusive privilege to add new revisions to the object history. This helps to make sure that no other author in your project, can interfere with your work.

If an object is accompanied by a file containing a specific version of the object, this file is called the *busy version*. The term shall indicate, that a busy version can be altered, e.g. by invoking a text editor on it, and thus isn’t a real version. The busy version is the interface between the whirling world of changes, and the stable world of secured versions in the archive. Technically, versions evolve by taking snapshots of the busy version.

You can retrieve revisions from the object base with the retrv command.

```
$ retrv xyzzy.c
xyzzy.c[1.0] -> xyzzy.c
```

fetches the latest revision of ‘‘xyzzy.c’’ from the object base and installs it in the current working directory. As this operation might overwrite a current busy version, retrv asks for permission to do so. In case you just want to have a look at the specified revision without overwriting the busy version, you should rather type

```
$ vcat xyzzy.c
```

than invoke retrv. The vcat command simply prints the contents of the revision to standard output. As is the case with vl, this command also tries to resemble regular UNIX commands that do similar things.

4. Preparing for a Modification

This is what version control is for. One of the most typical elementary processes in software development is the *reserve-{modify-build-test}*-deposit* cycle. The scenario is that a change request (CR), such as a bug report or design item, comes in and gets dispatched to a developer*. The developer analyzes the change request and eventually *reserves* a couple of source modules for modification. Depending on the complexity of the planned change, the programmer makes various attempts to implement the change, and thereby eventually creates numerous intermediate versions, mainly for possible backup. After all changes have successfully been tested, the developer *proposes* the new version(s) for approval as implementation of the CR. An important rule of version control is that one should always be able to tell the reason, *why* a particular version has been created. The approved version(s), for example, should be related to the change request that was implemented by them. Consequently, it is generally a good idea to get rid of all the intermediate versions after the new version(s) have been approved.

4.1. Reserving an Object

Just retrieving a revision from the object base is not sufficient if you wish to make a modification and store it as a new revision into the object base at a later time. Precisely, retrv in its plain form doesn’t set a lock on the object history, which is necessary in order to add a new revision to it. Accordingly, the access mode of the file containing the retrived revision is set to read only. If you wish to create a new revision on the basis of some saved version, you should retrieve it with

---

*) This is a rather simple (but very common) change control model. A more sophisticated model may ask the developer to prepare an analysis of the CR, and report it to a configuration control board (CCB) being in charge to decide whether, when, and how to respond to a given CR, before it is actually dispatched to the implementor.

```
$ retrv -lock xyzzy.c
Write access on xyzzy.c denied. Overwrite it anyway ? [no] yes
Write change intent for of xyzzy.c ? [yes] <CR>
<starting favorite editor>
xyzzy.c[1.0] -> xyzzy.c
```

If ‘‘xyzzy.c’’s object history is not locked by anybody else, retrv will ask for a description of the changes you plan to apply, and installs the retrieved revision as busy version in the directory containing the object base (this is necessary to keep the defined relationship between the busy version and the corresponding object history). The change intent you get promoted for should clearly state the reason, for a modification. Usually one has a much clearer idea of what the modification is about when a module is reserved than when it is checked back into the object base. A clearly stated intent is also a sign of progress (there are times when you want to know what’s going on in the project). If CRs are dispatched via electronic mail, it is often a good idea to use it in place of the change intent.

In the case that some other programmer has reserved the update privilege for ‘‘xyzzy.c’’ you would rather see:

```
$ retrv -lock xyzzy.c
xyzzy.c already locked by andy@cs.tu-berlin.de.
xyzzy.c not restored
...aborting
```

When this happens to you, you probably want to know what reason the other programmer had to lock the history (or if he maybe reserved the lock only to forget about it). The ‘‘vlog’’ command helps you to find out what’s going on:

```
$ vlog -intent xyzzy.c
xyzzy.c[1.0]:
  __Intent__=[Wed Aug 28 18:43:28 1991] Intention for change:
Correct spelling of odd message that comes up when
program is called with option "-foo".
```

It may happen from time to time that you applied changes to a (supposedly) busy version of a source object without holding a lock for it. The version control system would prevent you from submitting a new version into the history:

```
$ save xyzzy.c
xyzzy.c:
save - error: You must lock xyzzy.c[1.0] before saving.
xyzzy.c not saved.
...aborting
```

In this case you probably don’t want your changes to be overwritten by a newly installed busy version as would be created by ‘‘retrv –lock’’. Instead, you should issue the command

```
$ vadm -lock xyzzy.c
```

This command tries to set the necessary lock subsequently. If, however, some other author has locked the object history in the meantime, you are in limbo. Getting out of this dilemma is only possible by using human interaction. Locking assures that you, and only you, can store the next update of a source object. It helps to avoid nasty problems if several people are working on the same project, and possibly on the same source objects. Even if an object history is locked, revisions can still be retrieved for reading, compiling etc. All that locking prevents is a save operation by anybody but the locker.

5. The Scoop on Branching

In practical software development there are two common situations that frequently cause trouble for organized change control:

- emergency fixes to a module that has been reserved for modification by a developer who is temporarily unavailable
- regular (maintenance) fixes that have to be applied to old versions of a software system while the new release is already under development.

Both situations usually cause *branching* of the version history of one or more source objects. Arbitrary branching of version histories is found in many current source control systems. However, branching is a tricky issue that can lead to severe loss of control over the change process. Besides introducing complicated version identification schemes, branching often seduces developers to use it in the wrong place, namely to implement the concept of *variants* (variants are versions that are not related through the predecessor/successor relationship, but represent equivalent instantiations of the same concept, the *invariant* of several variants. There is no commonly agreed concept of variant other than this in the software engineering community.).

ShapeTools does not support explicit branching of version histories. The version control system keeps track of branches internally. Branches occur, if – for example – some older (than the most recent) version is restored for proceeding with development. This scenario is illustrated by the two figures below: the situation shown in Fig.2 would result from discarding the busy version in Fig.1, and subsequent restoration of ‘‘samprog.c[1.1]’’.

The basic version identification has only two simple levels: *generations* and *revisions*. The exact meaning of both is not precisely predefined but rather left for exact definition within the project policy framework. We recommend a concept that distinguishes ‘‘major releases’’ (represented by the generation number) and ‘‘minor releases’’ (revision number). The precise semantics of ‘‘major’’ and ‘‘minor’’ should be defined by appropriate procedures in a project.

ShapeTools’ simple version identification scheme typically maps to a temporal relation between the versions. I.e. if a version has higher version number it was (most likely) created after a version with a lower number (exceptions occur, see below). This is called the *logical* version history. Figure 1 below shows the logical version history of ‘‘samprog.c’’*.

---

*) The version graph figures have been automatically derived from the version archives: ShapeTools’ *vl* command extracts and formats the history information to produce a graph description. The graph description is then fed to *Edge*, an extensible graph editor written by Frances Newbery-Paulisch. The command that produces the version graph diagram on the workstation is a 15-line shell script.

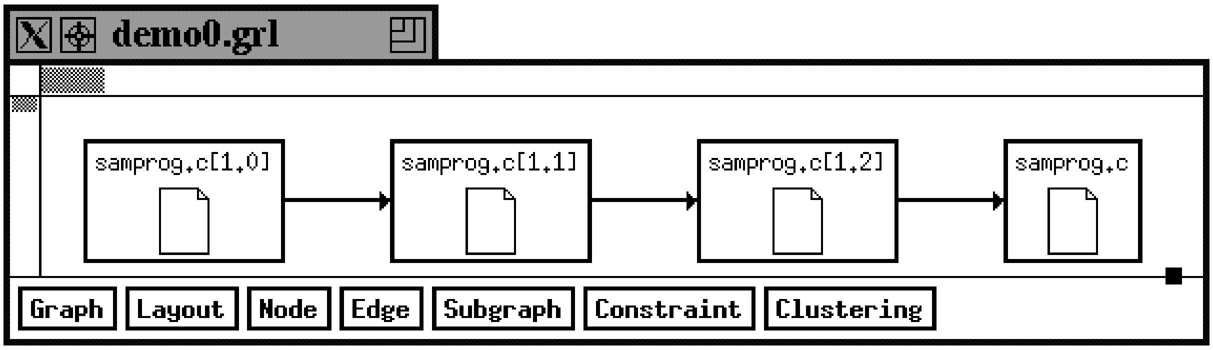

Fig.1: Logical Version History

Version numbers say nothing about the *physical* version history, describing which version is the result of modifying a particular other version. Logical histories are strictly linear, while physical histories include branches.
Figure 2 shows the ‘‘physical view’’ of the same objects history.

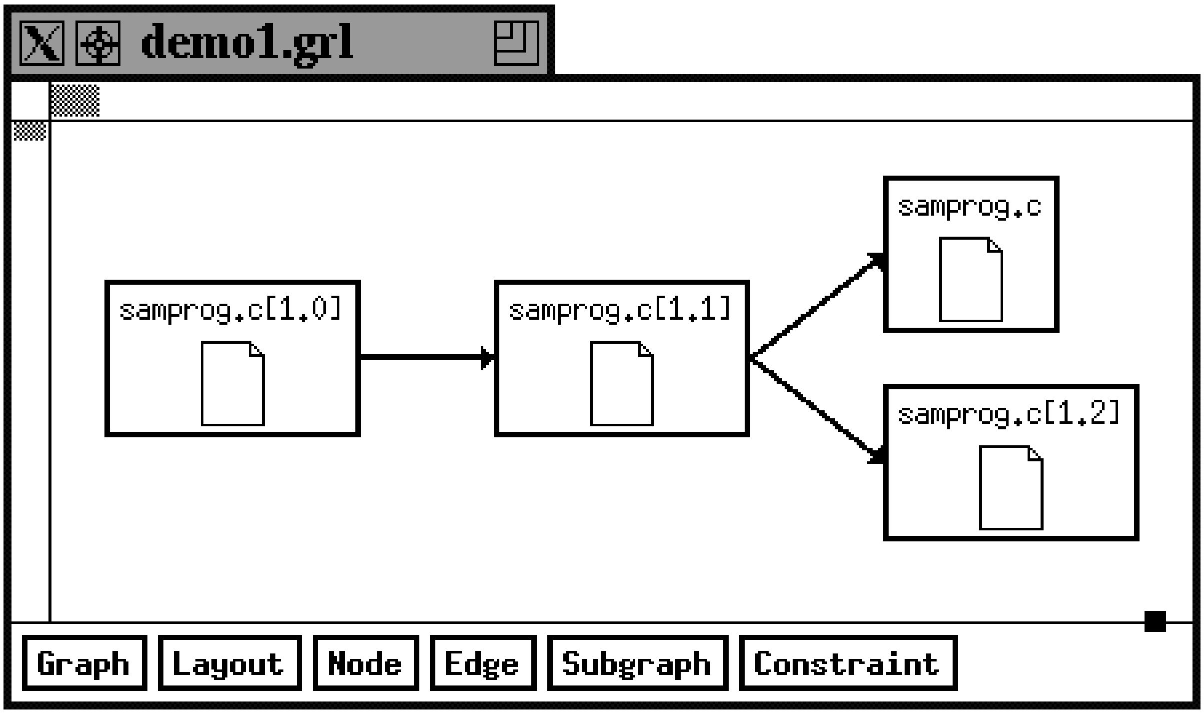

Fig.2: Physical Version History

Within the toolkit, version numbers are the most basic means of version administration. For the purpose of project specific revision and release control, it is much more suitable to use *symbolic names* as the problem- or organization-oriented version identification. It is rather straightforward to define a convention on how to construct symbolic names, and implement this convention through automated tools. We will provide an example in one of the later sections, describing a sample automatic release management system.

However, the two problems stated above are real problems that are very common. How does ShapeTools deal with them ?

5.1. Emergency Fixes

In case a modification *must* be applied, but the respective module(s) have already been reserved for modification by another developer, one probably sees some message(s) as this one:

```
$ retrv -lock scantif.c
scantif.c already locked by axel@cs.tu-berlin.de.
scantif.c[2.4] not restored
```

If the developer who currently owns the lock is unavailable, the lock can be *broken*. To break a lock means to unlock the object forcefully, thus making it available for locking by someone else. Only the project administrator (the user who ‘‘physically’’ owns the AtFS subdirectory where the version archives reside) can break a lock:

```
admin$ vadm -unlock scantif.c
scantif.c[2.4] currently locked by axel@cs.tu-berlin.de.
Break the lock ? [yes] <CR>
scantif.c[2.4] unlocked.
```

Usually, the previous owner of the lock will receive a mail message that warns him/her that the lock was broken, and reconciliation of concurrent modifications may be necessary.

5.2. Maintenance Fixes

It is a very common scenario that development of the next major system release proceeds while the ‘‘old’’ release which is currently operational, must be maintained. Maintenance means reacting to reported defects (bugs) or general modification requests (wishes). In practice, this problem often leads to numerous subtle problems that eventually will cause change control to collapse. Quite often, the solution will simply be to treat the old system release as a different product and drive maintenance just as it were development. The consequence is often that traceability of changes or even requirements (if at all present) will be seriously disrupted. Bug fixes that shall be propagated to the development area of the next release are often simply forgotten, causing the same bug that has already been fixed to be fixed again (and again...).

ShapeTools’ concept of *fixes* is based on the two-level version identification concept. Assume the following source history:

```
$ vl scantif.c
scantif.c[1.0]  scantif.c[1.7]  scantif.c[2.4]
scantif.c[1.4]  scantif.c[2.3]
```

The history is depicted by the following diagram

```
$ showhist scantif.c
...
```

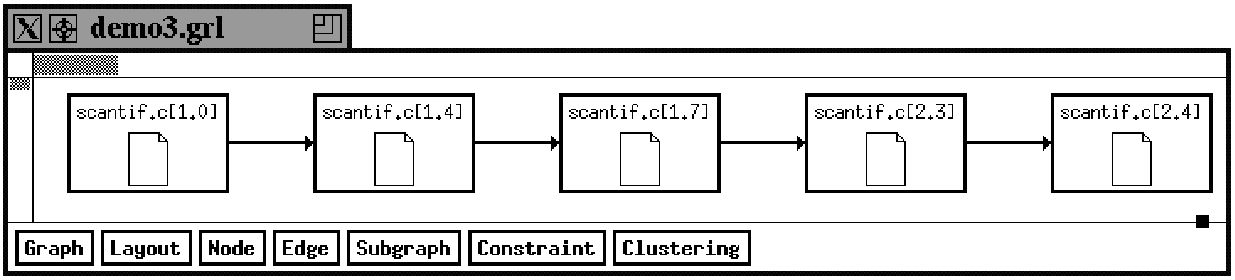

Fig.3: History before Maintenance Fix

Let's furthermore assume, that a client has reported a defect that led to the identification of a bug in "scantif.c[1.7]" which is part of the currently operational system release (major release level 1). The people in charge of developing the next release (major release level 2) are busy appending new versions at the end of "scantif.c"s history. They don't want to be bothered by peanut fixes to the old release. With ShapeTools, we can do the following:

```
$ retrv -fix scantif.c[1.7]
Describe intended changes of scantif.c ? [yes] <CR>
< Fix this "peanut" bug in operational release >
scantif.c[1.7] -> scantif.c
```

IMPORTANT: Note that we should have created a separate maintenance workspace that must be connected to the projects source repository. This is as much work as typing "*mkdir fix; cd fix; ln -s <project-path>/Atfs.*" (see also below). Now, the bug can be fixed without interfering with the development team. See the sections on *shape* to find out, how system-build and unit test can be performed without interfering with the rest of the project.

When a fix has been tested and approved, it is time to check it into the version archive. This must be done with the following *save* command which is symmetric to the module reservation above:

```
$ save -fix 1 scantif.c
scantif.c:
Do you want to comment your modifications ? [yes] <CR>
< ...was more sort of a coconut !! >
scantif.c[1.8] saved.
scantif.c removed.
```

The save command above should read "save the fix that is in file scantif.c and append it as most recent revision in major release level 1 of scantif.c." The diagram in figure 4 shows the resulting physical version tree:

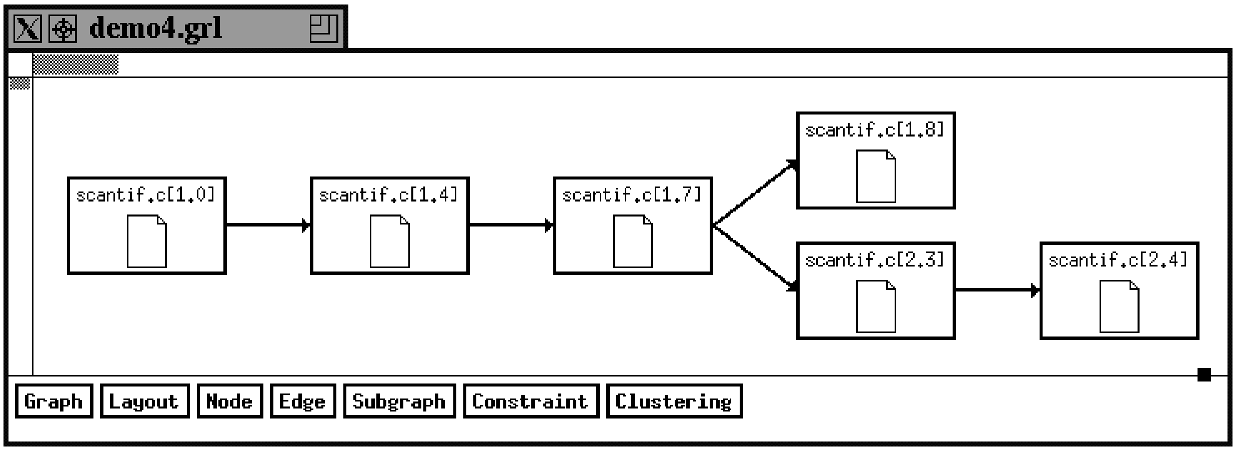

Fig.4: History after Maintenance Fix

ShapeTools' fix mechanism is easy to handle if it is well coordinated with the version numbering policy. In our example, we assume that generation 1 of our system represents the operational release, while generation 2 of the system is still under development. Once release 2 is completed, a new generation will be initiated in order to keep release 2 maintainable.

Extracting the fix to generation 1 of the system is easy with ShapeTools:

```
$ vdiff -c scantif.c[1.7] scantif.c[1.8] | mail rel2-dvlpmnt
```

5.3. Merging a Fix into the Main Line of Development

It is often the case that a bugfix to the maintained system configuration is also relevant for the development configuration. In the example above, a so called *context diff* – a list of differences between the original and the fixed version (along with some textual context around the changed lines) – is extracted and mailed to the development team for evaluation. If the development team finds that the bug which has been fixed in the previous system release is still present in the upcoming release under development, they will have to apply this fix to the main development version.

Re-applying a known fix manually is a tedious and error prone undertaking. Re-applying a fix automatically is much more convenient and usually a lot safer than to do it by hand. The process of automatically incorporating a change (such as a bug fix) into a different version is often called *merging*. For merging to work, a fix (also often called *patch*) must be in a certain format that can be interpreted by an appropriate tool. Examples of such formats are command scripts for batch oriented text editors, such as *ed*. A newer, very common format for patches is the *context diff*, as produced by the UNIX command *diff −c*. Use the *patch* program (written by Larry Wall) to merge a fix into a version under development.

5.3.1. Applying a Patch

Before a patch is applied to a source object, care should be taken that the intended change can be precisely isolated, among other reasons to have the option to revert the effect of the change, just in case. Thus, the recommended procedure for applying a patch is:

- save the current state of the source object about to be patched
- apply the patch by invoking the patch processor

- save the the state of the source object directly after the patch and give it an appropriate name to have the patched version identifyable.

The difference between the two versions that have been created in the process should be equivalent to the patch that has been applied. A sample session following the above strategy is shown in the box below.

```
$ save -lock scantif.c
scantif.c:
Do you want to comment your modifications ? [yes]
<Safepoint before patch is applied>
scantif.c[2.5] saved.
$ patch scantif.c < patch-0815
Hmm...  Looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** scantif.c [1.7]       Wed Feb 12 21:15:45 1992
|--- scantif.c [1.8]       Mon Mar 23 12:51:09 1992
--------------------------
Patching file scantif.c using Plan A...
Hunk #1 succeeded at 695.
done
$ save -lock -q -alias patch-0815 scantif.c
scantif.c[2.6] saved.
```

The version graph that results from the merger above is shown in figure 5:

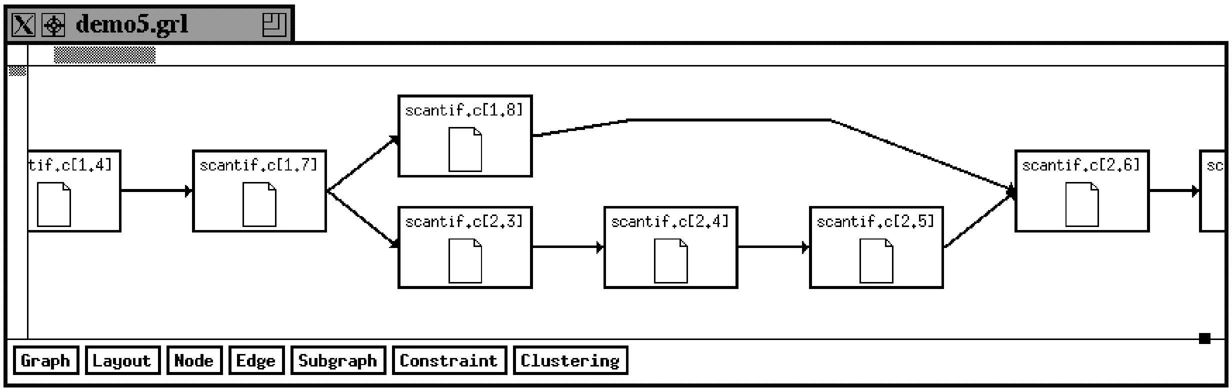

Fig.5: Version History After Applied Patch

5.3.2. Automatic Merging is not Safe

Automatic patching is a very efficient and convenient way to propagate changes and bug fixes in various situations occuring in the process of software development. Text based merging, however, is not semantically safe. Text based patch processors are nothing but text processors, and all they can do is to transform one piece of test into another piece. These tools have no idea whether the text they are transforming is a memo, a manual page or a program module in some programming language. It is very well possible that for example the invariant of a procedure or variable has been changed inappropriately by a patch. Thus, the automatic incorporation of a bug fix may cause new bugs that may show up only at runtime. To avoid this situation, patches should be reviewed carefully before they are applied. Version control is a powerful instrument to keep the overall situation under control.

Despite the possible problems associated with automatic merging, experience shows that the concept works very well in practice. Real problems occur only rarely.

6. Proceeding with Regular Development

Once the update privileges are reserved for a set of source modules, the implementation and unit-test phase begins. During this phase, a developer usually saves a number of private revisions for possible backup. The resulting version objects in the object base are automatically marked with the status *saved*, indicating that they are not intended for public access. Note, that the status does *not restrict* access to versions but is of advisory nature.

After a couple of development and test iterations (precisely: *think-modify-build-test-..*) a CR will eventually be satisfied. If this point is reached, the developer will *submit* some version of all modified source objects for inclusion into the next project baseline. How this is done will be shown shortly.

6.1. Referencing Versions

After having used version control commands for a while, you might have quite a lot of revisions in each object history. The vl command could print something like

```
efg.c          fgh.c          fgh.c[1.3]   xyz.c[1.1]
efg.c[1.0]     fgh.c[1.0]     fgh.c[1.4]   xyz.c[1.2]
efg.c[1.1]     fgh.c[1.1]     xyz.c        xyzzy.c
efg.c[1.2]     fgh.c[1.2]     xyz.c[1.0]   xyzzy.c[1.0]
```

While most version control commands access by default the most recent revision, it is possible to reference any revision by explicitly specifying its revision number. This can be done in either of two ways common to all of ShapeTools' version control commands. The command option -V<version> specifies an explicit default revision number. For example, the command:

```
$ vl -l -bind 1.2
```

prints more detailed information about revision 1.2 of all object histories:

```
-rw-r--r-- s axel@cs.tu-berlin.de       437 Nov  2 01:21:46 1988 efg.c[1.2]
-rw-r--r-- P axel@cs.tu-berlin.de      1027 Oct 26 22:04:33 1988 fgh.c[1.2]
-rw-r--r-- s axel@cs.tu-berlin.de       731 Nov  1 02:01:34 1988 xyz.c[1.2]
```

The other notational convention understood by all version control commands is the bound version notation. The revision number is simply specified as an extension to the object name, enclosed in brackets. The command

```
$ vl -l \*[1.2]
```

will issue the same output as above. Note that in the astrisk has to be escaped as it shall be expanded by the vl command rather than by the shell. Another example, the command:

```
$ vl -l efg.c[1.2] fgh.c[1.1] xyz.c[1.3]
```

prints

```
-rw-r--r-- s axel@cs.tu-berlin.de       437 Nov  2 01:21:46 1988 efg.c[1.2]
-rw-r--r-- s axel@cs.tu-berlin.de       726 Oct 26 22:04:30 1988 fgh.c[1.1]
-rw-r--r-- s axel@cs.tu-berlin.de       727 Nov  1 02:25:51 1988 xyz.c[1.3]
```

6.2. Submitting a Version

Now that it's clear how particular versions are referenced, it is easy to submit specific versions for inclusion into the next product baseline:

```
$ sbmt xyzzy.c[1.4]
Status of xyzzy.c[1.4] set to 'proposed'.
```

The version submission procedure described in this example is rather simple. The developer even has the freedom to declare his newly submitted version public or even frozen (see below). Depending on how a particular project works, one may want to have intermediate administrative steps to assert certain quality criteria. Wrapping the different status sensitive operations into a script provides a simple solution to this problem. Note, however, that ShapeTools does not provide for any kind of mandatory access control.

The resulting state of the version archive looks like:

```
$ vl -l xyzzy.c
-r--r--r-- P axel@cs.tu-berlin.de    1866 Aug 12 15:35:28 1991 xyzzy.c[1.0]
-rw-r--r-- s axel@cs.tu-berlin.de    2010 Dec  4 11:42:13 1991 xyzzy.c[1.1]
-rw-r--r-- s axel@cs.tu-berlin.de    2291 Dec  9 20:06:23 1991 xyzzy.c[1.2]
-rw-r--r-- s axel@cs.tu-berlin.de    2766 Dec 20 14:22:04 1991 xyzzy.c[1.3]
-rw-r--r-- p axel@cs.tu-berlin.de    2438 Jan  3 18:42:24 1992 xyzzy.c[1.4]
```

The long format of the version list command is similar to ls (1) long format file information. The information printed in this format is (from left to right): file access permissions of the busy version (when saved), revision status (b = busy, s = saved, p = proposed, P = published, a = accessed, f = frozen), network userid of the owner of the object history, original size of the revision (revsions are actually stored as differences to the previous revision), date and time when the revision was saved, and name, type (represented as name suffix), generation-, and revision number (to either side of the dot). All these informations are examples for individual attributes of version objects. Many other attributes are stored for each version object that can be examined by vl (try vl -all to see all of an objects or files attributes).

As can be seen in the listing above, there are now five versions of xyzzy.c. While the first (xyzzy.c[1.0]) and the most recent version (xyzzy.c[1.4]) are either in published or proposed state, the other versions are obviously intermediate. In order to keep the pool of versions in the object base small (and reports from the object base instructive) it is often a good idea to delete intermediate objects. This can be done with ShapeTools' general purpose version administration command *vadm*:

```
$ vadm -delete xyzzy.c[1.1] xyzzy.c[1.2] xyzzy.c[1.3]
Delete xyzzy.c[1.3] ? [yes] <CR>
xyzzy.c[1.3] deleted.
Delete xyzzy.c[1.2] ? [yes] <CR>
xyzzy.c[1.2] deleted.
Delete xyzzy.c[1.1] ? [yes] <CR>
```

```
xyzzy.c[1.1] deleted.
done.
```

The version control system makes sure that only intermediate versions (with status saved) can be deleted. There is a short form for *vadm -delete* which reads:

```
$ vrm xyzzy.c[1.1] xyzzy.c[1.2] xyzzy.c[1.3]
...
```

ShapeTools has been designed to blend smoothly with the Unix standard tool concept. To illustrate how this works, we can delete those intermediate versions in a slightly different way:

```
$ vadm -delete `vl -s saved xyzzy.c`
Delete xyzzy.c[1.3] ? [yes] <CR>
xyzzy.c[1.3] deleted.
Delete xyzzy.c[1.2] ? [yes] <CR>
xyzzy.c[1.2] deleted.
Delete xyzzy.c[1.1] ? [yes] <CR>
xyzzy.c[1.1] deleted.
done.
```

The construction in this example is a *shell-substitution*, i.e. the command enclosed by the grave accents is executed first, and its output is substituted for it†. The command

```
$ vl -attr state=saved xyzzy.c
```

produces a list of all xyzzy.c-versions that are in state saved. This list is then passed as argument to vadm. This way of using the vl command is very useful to apply certain commands on groups of versions. As will be shown below, vl provides a lot of flexibility, to generate various reports from the object base.

7. Finding out more about Objects

The version-list program vl can be used in many interesting ways. When called this way:

```
$ vl -last
Dependencies[1.1]    doretrv.c[3.28]      retrv.h[3.6]
Makefile[1.17]       retrv.1[3.9]         retrversion.c[3.56]
Shapefile[1.8]       retrv.c[3.21]
```

vl displays the identifiers of all the most recent revisions in the current version object base (the AtFS subdirectory). This usage of vl is so common that we already considered to make it *vl*s default behavior.

In more ambitious project settings that involve somewhat more sophisticated configuration management functions, *vl*s ability to use symbolic names instead of version numbers is of great utility. Assuming the existence of certain (automatically supported) naming conventions for configurations, you can list all versions that are part of a configuration by typing:

```
$ vl -bind ShapeTools-1.2
Dependencies[1.2]    dosave.c[3.48]       save.h[3.13]
```

† If you are using the C-Shell (or tcsh), you must "set noglob" to keep csh from complaining about "no match".

```
Makefile[1.18]       save.1[3.8]        saveversion.c[3.56]
Shapefile[1.10]      save.c[3.29]
```

For more complex products with lots of subsystems that span over several directories, the command can also be applied with an additional recursion switch:

```
$ vl -R -bind ShapeTools-1.2
<output omitted>
```

To see how easy it is to generate object base reports in custom formats, check the section "Using Attributes".

7.1. Keeping Track of Configurations

As already illustrated by the previous section, keeping track of configurations, composed of particular versions of many different objects, is easy with the ShapeTools version control system. While the ultimate tool to create, maintain, and operate with configurations in this toolkit is the shape program, the version control system allows to associate different objects in a configuration, and is able to reinstall a given configuration from the object base with one single command. Saving a simple configuration, consisting of all the current instances of the busy versions can be done by invoking

```
$ save -alias conf1 efg.c fgh.c xyzzy.c \
      -logmsg "versions work together"
```

Provided all necessary locks are (or can be) set, save will print something like:

```
efg.c:
efg.c[1.3] saved.
efg.c removed.
fgh.c:
fgh.c[1.5] saved.
fgh.c removed.
xyzzy.c:
xyzzy.c[1.1] saved.
xyzzy.c removed.
done.
```

Each newly created version is assigned the name of the configuration as symbolic name attribute (n option). The text supplied with the m (short for message) is taken as log entry for the evolving versions, so save won't prompt for a descriptive log. The specified symbolic name is an alias for the version number. No single symbolic name may be assigned to more than one version in an object history. A single version may be assigned any number of symbolic names, however. The symbolic name of a version may be used anyplace where a version number might be used. So, an entire configuration can be reinstalled with the command:

```
$ retrv -bind conf1 efg.c fgh.c xyzzy.c
```

An alternative way to reinstall an identified configuration that is especially useful for larger systems (i.e. when you want to avoid typing all those object names):

```
$ retrv `vl -bind conf1`
```

8. Determining Differences between Versions

ShapeTools provides another Unix-ish tool to determine the differences between two versions, the command *vdiff*. Vdiff is not a core part of the ShapeTools system but rather an application of the toolkit idea. It is a shell-script that employs ShapeTools commands and the Unix *diff*-command. If you are a shell-programmer, you are encouraged to look into the code of this program.

Vdiff ‘‘understands’’ almost all options of your local *diff* program - they are simply passed to it. The command interface however, complies with the ShapeTools-standard, i.e. you can pass arguments in bound version notation.

One of *vdiff*s most convenient properties is that it can be used to generate upgrade patches from one module- or system release to another. This is typically done by commands that look like:

```
$ vdiff -c doretrv.c[ShapeTools-1.1] doretrv.c[ShapeTools-1.2]
```

How this basic command can be extended to generate complex patches for entire system releases is left as an exercise to the reader.

9. Identification of Object Code

It is good practice to insert a piece of information into the source code of program modules that will allow to identify the source object versions, a given piece of object code has been compiled from. In C programs for example, such a piece of information typically looks sort of

```
static char *RCSID =
"$Header: vcintro.1,v 3.0 88/11/09 23:12:21 axel Exp $";
```

or

```
static char *SCCSID = "@(#)xyzzy.c 1.2  8/11/88";
```

The two examples displayed above show the standard identification string format used by the RCS and SCCS source control systems respectively. The cryptic character sequences serve as magic cookies for special identification programs which extract these strings from the compiled objects or executable programs (ident and what). ShapeTools’ version control system has some special support for this kind of code identification. If you insert the line

```
static char *AtFSID = "$ __Header$";
```

in the source of the program modules, the version control system will expand this expression into

```
static char *AtFSID=
 "$Header: save.c[3.24] Thu Dec 28 19:44:24 1989 axel@coma published$";
```

allowing to identify compiled objects with RCS’ ident command. The subsequent section on attribute citations will show how to design your own source identification string.

10. Attribute Substitution

The sequence ‘‘$_ _Header$’’ is an attribute citation expression. When a version is retrieved from the object base, attribute citations in its contents are substituted by the value of the cited attribute. In the given example, the magic string "$_ _" indicates the beginning of an attribute citation. The string that immediately follows the citation marker is the name of an attribute. If the object has such an attribute and the attributes value is defined, it will be substituted in place of the attribute citation expression. An attributes value is not necessarily some literal but can also be a command execution (see below).

Each object in the object base has a number of standard attributes and may additionally have any number of so called user defined attributes (see below). A "$" character or white space immediately following the attribute name delimits an attribute citation. If no attribute with the specified name is associated with an object version, the whole citation will be ignored, i.e. treated as ordinary text. Attribute citation can be used quite flexibly as a macro or include mechanism.

"Header" is one of two pseudo attributes that each version object has in addition to its regular attributes. It expands into a string that displays a number of a versions attributes all at once. The "Log" pseudo-attribute pops in the complete log history of the respective version. This is useful to add some kind of development documentation to a particular version. To prevent the inserted log text from causing syntax errors, each inserted line is preceded by a comment leader symbol, which is stored as yet another attribute. In order to use this feature properly, the comment leader symbol must be explicitly set for each source object that has a syntax requiring comments to be marked. For C language objects, this can be done with

```
$ vadm -setc " * " xyzzy.c
```

It is a good idea to use a utility program, such as afsit, to insert proper object identification and log attributes automatically. The lines that afsit adds to C files look like

```
static char *AtFSid = "$__Header$"
/*
 * $__Log
 */
```

11. More on Attribute Citations

Creative use of ShapeTools attribute citation gives the advanced user a lot of flexibility. For example, a source line that is equivalent to the "Header" attribute but uses only standard attributes looks like:

```
$Header: $__self$ $__stime$ $__author$ $__state$$
```

Composing a source identification that displays other attributes of the source version is trivial. All one has to know are the names of the attributes. For a list of all standard attributes and their meanings, check Appendix A.

Another exciting application of attribute citations is the "–format" option of the *vl* command. Produce a custom format report about the versions in the object base by composing a report line pattern with attribute citations just as in the "Header" example above. A format specification that (approximately) matches the output of "vl -l" looks as follows:

```
$ vl -format '$__mode $__state $__author $__size \t \
              $__mtime $__self$\n'
```

The quoting is needed to keep the shell from messing around with the money. If one wants the report to be a little more verbose, maybe

```
$ vl -format 'Object name: $__name$. The objects \
             type is $__type$.\nVersion $__version was written \
             by $__author$,\nand checked in on $__stime$.\n\nThe \
             net change with respect to the preceding version\n \
             ($__pred$) is rated $__dsize$.\n\n' -last xyzzy.c
```

is a suitable choice. *vl*s format option offers a great deal of flexibility, especially if it is used in scripts (or perhaps in an environment) that pass the output to other tools (e.g. *awk*) in order to produce even more elaborate reports.

12. Using Attributes

So far, we have only seen some of ShapeTools' built-in standard attributes. The concept of *user-defined attributes*, however, is one of ShapeTools' most remarkable features. It was introduced during the design of the version control system mainly because we weren't sure about "how many (or what) attributes are sufficient for a version control system". As a consequence we decided to keep that matter open and introduced a mechanism that provides for an arbitrary number of attributes with arbitrary length.

Since then we used this attribution almost every time we introduced a new concept or feature into the system (e.g. change intents). User-defined attributes are also very useful when the ShapeTools toolkit is used to build somewhat more complex tool environments. Attributes of software objects are ideally suited to *communicate information between unrelated tools* within an environment. All what is needed is typically a little wrapper shell-script.

User-defined attributes can be attached to versions as well as to regular UNIX files that don't have any versions (you need to have an AtFS subdirectory though). Attributes are attached by

```
$ vadm -attr last_release="conf1-rel4pl2" xyzzy.c
```

You can use such an attribute to identify objects that bear the attribute or bear that attribute with a certain value:

```
$ vl -l -attr last_release=conf1-rel4pl2
-rw-r--r-- b axel@cs.tu-berlin.de 10 Aug 28 19:18:46 1991 xyzzy.c
```

Attributes can be used in many different ways. The command **vattr** (short for "vadm -attr") allows read and write access to attribute from the UNIX command level). *vattr* is ver useful when programming ShapeTools shell-scripts. Examples for reading

```
$ vattr mtime xyzzy.c[1.0]
Mon Jun 21 11:57:05 1993
```

```
#!/bin/sh
t1=`vattr mtime xyzzy.c[1.0]`
t2=`vattr mtime xyzzy.c`
if [ $t1 -eq $t2 ]
then
  echo xyzzy.c[1.0] and xyzzy.c are identical.
fi
```

and for setting attributes with *vattr*:

```
$ vattr release_name=internal_prerelease-6 xyzzy.c[1.0]

$ vattr release_name=+official_release-1.0 xyzzy.c[1.0]
```

The =+ in the last example causes vattr to *add* the value to an eventually existing list of attribute values. Just giving an equal sign causes any existing values to be overwritten.

Conceptually, there is no difference between built-in, and user-defined attributes. Both can be used in very much the same way. Values of standard and user-defined attributes may not only be cited within format specifications or the contents of an object, but also in attribute values itself. Attribute citations are recursively evaluated. A simplified version of the built-in attribute *self* could thus be defined in the following way:

```
self=$__name$.$__type$[$__version$]
version=$__generation$.$__revision
```

13. Different Kinds of Attributes

There are three basic kinds of attribute values: genuine values, reference values, and execution values. Genuine values are simply strings that are assigned to an attribute. Reference values are pointers to files or AtFS-versions whose contents will be substituted in place of an attribute-citation. Reference values are strings that begin with a circumflex-character, typically followed by pathname, e.g. ‘‘^/usr/local/lib/std-header[2.4]’’. Execution values are names of executable programs, whose standard output is substituted in place of an attribute-citation. Execution values are strings that begin with an exclamation-mark character, typically followed by the name of the program, e.g. ‘‘!/bin/date’’. Execution values can be used to generate highly dynamic attributes or a primitive form of event-triggers.

One of our favorite applications of execution value attributes are ‘‘talking documents’’. On modern workstations that support recording and playing of audio data, it is easy to place attribute citations at appropriate positions within a document and associate ‘‘play sound’’ commands to these attributes.

```
$ vattr sound1=blurb1.au xyzzy.c
$ vattr speak1='!play $__sound1$'
<... more sound attributes set>
$ vcat xyzzy.c
```

Make sure to cite the speak attributes at appropriate places within the talking document. This is especially effective if you switch your terminal window to smooth scrolling. Have fun!

Execution values are not just another toy-idea, however. One of the more serious applications would be to associate functions that perform all sorts of analyses on the various source versions of a product. If these analyses provide exact data (such as size or complexity measures), this data could - e.g. through use of ‘‘vl -format’’ - be used to generate statistics. We believe that attributes provide an ideal basis for the introduction of quantitative methods (after all, every quantification results in a value that can be represented as an attribute).

14. Setting up a Project

Before concluding the introductory discussion of ShapeTools version control facilities, it shall be described what we think is a good organizational scheme for setting up multi-programmer development projects. A few basic assumptions are built into the system itself. However, the proposed scheme is in no way a prerequisite for efficient use of the toolkit. Later, in the part about the *shape* program, the described approach to project organization will be illustrated by an example project.

Within a project, there are three basic roles that a programmer may play with respect to an object: *owner* of the entire design object (i.e. all of its versions), *author* of a particular version, and *locker* of the object history. All three roles may be played by different people for each object. The network userids of the people who play these roles are recorded as attributes in the object base. The roles will be reflected in the organization of the *source repository* and the local workspaces.

14.1. Repositories and Workspaces

First, a central place for all of the projects components must be created, the *repository*. Once the repository has been established, it is trivial for a developer to create a workspace and connect to the repository.

Before establishing the project repository, create a designated userid to be the *project owner*. The project owner will be the owner of all objects that are part of the project and acts as administrator. Create a special entry for the project - the *project group* - in the /etc/group file. The central object base for a project will be installed as an AtFS subdirectory in the project owners home directory. The AtFS directory should have the group ID of the project group, and should be group-writable. The project owners home directory will also be referred to as project home. The project home should be the integration area, i.e. the place, where the published work results of the various team members are collected and integrated into the product that is subject of the project.

All programmers who are member of the project team should create a special directory that is dedicated to the project. To connect this workspace directory to the project repository, the programmer only needs to create a symbolic link called AtFS that points to the AtFS subdirectory in the project home. In SystemV based environments, create a file called AtFS that contains the absolute pathname of the project homes AtFS directory. By using this mechanism, all team members share the same object base. It should be made sure that each team member (also called author) is member of the project group (the UNIX group). This is necessary in order to grant all authors write access to the object base.

A sample setup for a project using a central repository and two separate workspaces for developers is shown in figure 6.

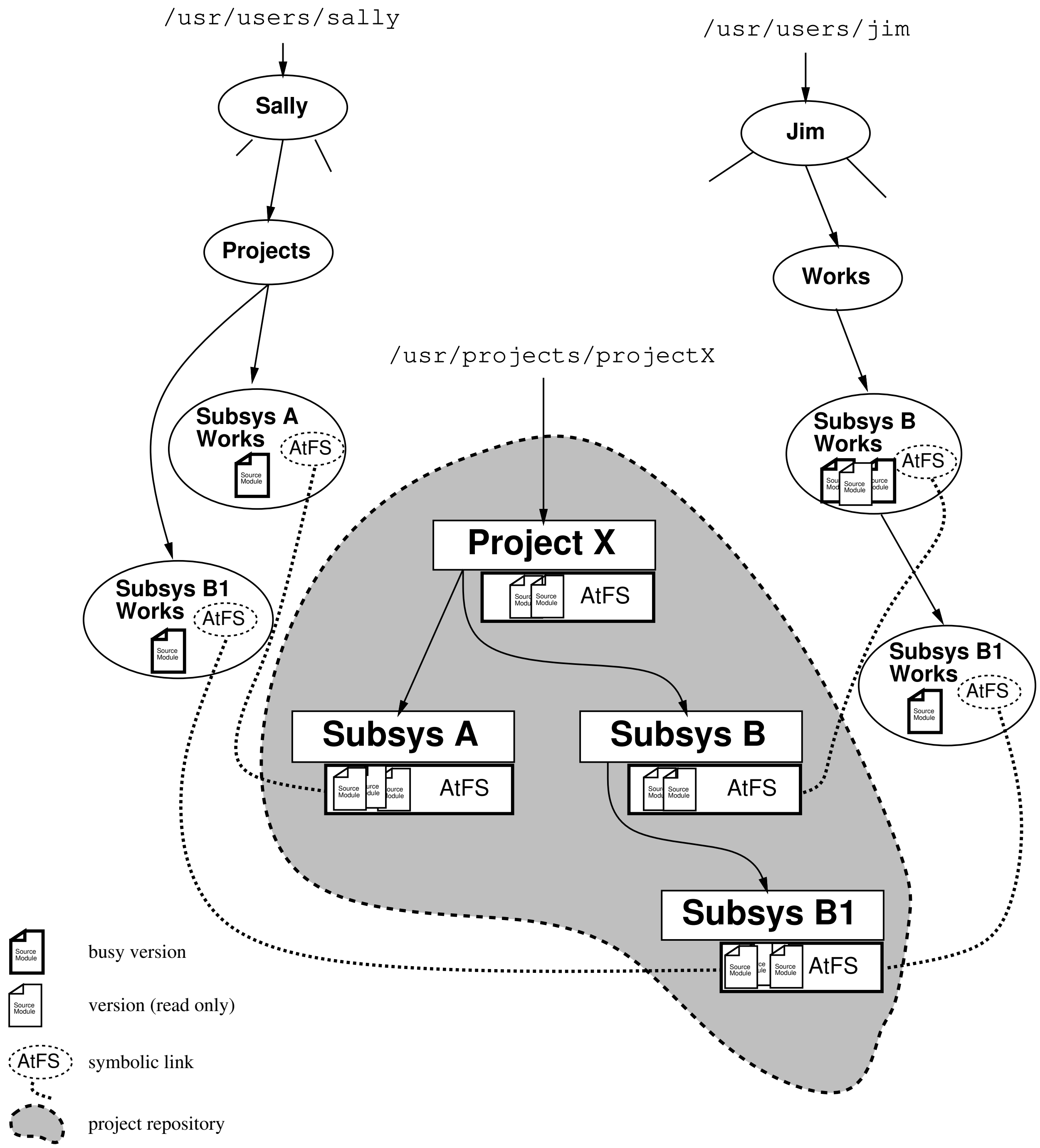

Fig.6: Project Repository and Workspaces

Accessing and selecting particular versions in a dynamic but well controlled and organized fashion is best be done with the shape tool. Basic version control can only supply the very basic means for sophisticated technical project support. Particular management policies for projects require more effort. However, the proposed scheme has been proven to work well for medium sized projects.

The subsequent sections will provide a synoptical description of *shape*s capabilities and explore them in a number of practical scenarios.

PART II — TEAM ORIENTED DEVELOPMENT WITH SHAPE

The *shape* program is the largest and most complex single element of the shapeTools toolkit. Its main purpose is to provide and maintain control over the evolution of the overall component structure of a potentially complex software system. In particular, shape provides fine-grain control over the build-processes that occur during the course of a development project. In team development settings, build requirements vary substantially as the project proceeds through numerous phases. In this tutorial, we will consider requirements for builds during individual component development and unit testing, as well as integrative builds for system integration.

Although shape's most obvious characteristic is that of a sophisticated build-tool, its scope of functionality goes far beyond builds. Among the more advanced applications of shape is a complete release management system which will be described in some depth later in this tutorial. As the example of the release management system will show, shape (and the associated system description files) provides a natural center around which product-oriented procedures of the software development process can be arranged. Furthermore, the simple, hierarchical syntax of shape's description files (the *Shapefile*s) is surprisingly well suited to describe many aspects of the overall software development process.

Despite the broad spectrum of functionality provided by shape, it is still easy to control. The basic simplicity of the tool is inherited from Make which was the main inspiration for the design of shape. Shape tries to build upon well known concepts (and working experience) to the highest possible degree. Thus, shape's control files, the *Shapefile*s, offer the same syntactic elements as Makefiles. The way shape is used resembles strongly the way one uses make.

For a thorough understanding of shape, it will be of great help if the reader of this tutorial is familiar with make, and with writing (or at least understanding) Makefiles. It is also recommended to read the shapeTools version control tutorial first. The tutorial starts with an outline of shape's most important capabilities. It will be shown, how shape is used for component development and testing. Once you have a feeling for the way shape functions, it will be easy to understand how this functionality is controlled. In subsequent sections, the elements that distinguish Shapefiles from Makefiles will be introduced. How these *rules* are applied to drive a well controlled system integration process will be described thereafter. The scenario will be extended to demonstrate how the common problem of maintaining different *variants* of a system is addressed with shape. While these aspects are discussed, it will continuously elaborated how to write good Shapefiles. The tutorial concludes with a description of the release management system that is used to control the evolution of the shapeTools toolkit itself.

15. Functional Overview of Shape

Efficient program builds from version repositories. Shape allows to transparently compile source objects that are either regular files in the current directory or *versions* of a source object, living in an archive file. Shape keeps track of the relevant parameters for compilations (source versions, compiler versions, compiler switches etc.) and thus provides a safe and efficient build machinery.

Derived object management and cooperative builds. When shape compiles source objects, it stores the resulting *derived objects* together with the effective compile parameters in its *derived object cache*. In team development it is frequently the case that different versions of sources are processed in parallel. Caching and restoring of objects that are derived from immutable versions allows developers to profit from previous builds by other team members. Overall compile costs in projects are substantially reduced.

Dynamic binding of versions to object names. When serving a build request, shape has to consider a possibly large number of versions that are stored for an object. Which particular version is *bound* to an object's name in the Shapefile, is determined by *version selection rules*.

Dynamic macro redefinitions. Most − if not all − of the objects and parameters that are involved in a build request, are defined as *macros*. Shape allows to dynamically alter some or all of these macros, depending on which target is pursued.

Variant management. Shape can serve build requests for different variants of a system in parallel. While most SCM systems that address this problem are implementing this concept with *branches* in the version history (version *tree*), shape uses the combined potential of dynamic version selection, dynamic macro redefinition, and derived object management.

System structure definition. To perform its task, shape must be provided with structural information about the system to be managed. This information is stored in the Shapefile. Because of its very nature, the Shapefile is an ideal information base for all sorts of product related definitions. Shape encourages the development of a set of (project− or organization-specific) conventions for system description, and provides a simple way to extract this information for use by other tools.

Declarative language to define software processes. The Shapefile syntax not only serves to specify component dependencies that are relevant for build processes, but allows a general, hierarchical definition of product oriented tasks. The concept of recursive dependencies maps directly to a stepwise refinement of task definitions. Such tasks can be fully, partly, or not at all automated as appropriate. Thus, certain activities may be automated and standardized, while other activities are just informally described in order to document them or to reason about them.

16. Organizing Cooperative Development

Before we dive into the specifics of using shape, let's assume a simple scenario for a cooperative software development project:

Jim and Sally are participating in a project to develop and maintain the program ‘‘samprog’’. The program consists of three source code modules (‘‘tif2ps.c’’, ‘‘scantif.c’’, and ‘‘genps.c’’), and two header files (‘‘defs.h’’, and ‘‘tif2ps.h’’). The file ‘‘version.c’’ has the only purpose to carry the version number for the entire program. ‘‘Shapefile’’ is the control file for the sample system. All these objects are under configuration control and *owned* by the project's source administrator ‘‘samprog_admin’’. They are stored (along with all of their versions) in a central source tree (*repository*), ‘‘/usr/projects/samprog’’. This is the ‘‘physical’’ location of the source module's *version archives* which reside in a subdirectory ‘‘/usr/projects/samprog/AtFS’’.

Our two programmers use dedicated directories in their private workspaces for performing their development tasks, lets say ‘‘/usr/users/jim/works/samprog’’ and ‘‘/usr/users/sally/samprog’’.

16.1. Accessing the Source Repository from Private Workspaces

Before Jim and Sally can apply any changes to the samprog sources, they must ‘‘connect’’ to the source repository. This is done by creating a *symbolic link* from the workspace to the source repositories' AtFS subdirectory:

```
jim> cd works/samprog; mkatfs -l /usr/projects/samprog/AtFS
AtFS repository linked.
```

For the purpose of this example, we shall assume that samprog has already been developed to a certain stage, and will now undergo continuous enhancements, carried out by the programmer team. The current state of the source repository is shown below:

```
jim> vl
AtFS              genps.c[1.1] tif2ps.c[1.2]
Shapefile[1.0]    scantif.c[1.0]    version.c[1.0]
Shapefile[1.1]    tif.h[1.0]        version.c[1.1]
defs.h[1.0]       tif2ps.c[1.0]     version.c[1.2]
genps.c[1.0]      tif2ps.c[1.1]
```

Once the repository link has been established, the programmers can build samprog:

```
jim> shape
shape - executing: cc -O -DUNIX -DBIG_ENDIAN  -c tif2ps.c
shape - executing: cc -O -DUNIX -DBIG_ENDIAN  -c scantif.c
shape - executing: cc -O -DUNIX -DBIG_ENDIAN  -c genps.c
shape - executing: cc -O -DUNIX -DBIG_ENDIAN  -c version.c
shape - executing: cc -o samprog tif2ps.o scantif.o genps.o version.o
```

This build has been completely from the archive: When no regular files are present in the working directory, shape tries to retrieve the Shapefile and all modules referenced therein from the version archive*. By default, shape selects the most recent version of all involved objects. The Shapefile that guided the operation is rather simple:

```
jim> vcat Shapefile
#
# Shapefile for ``samprog''
#
CFLAGS = -O -DUNIX -DBIG_ENDIAN
PROGRAM = samprog
SOURCES = tif2ps.c scantif.c genps.c
INCLUDE = tif.h defs.h
VERSION = version
OBJECTS = $(SOURCES:.c=.o) $(VERSION).o

$(PROGRAM): $(OBJECTS)
        $(CC) -o $@ $(OBJECTS)

$(OBJECTS): $(INCLUDE)
```

The Shapefile in the above figure is actually a vanilla Makefile, i.e. none of the distinctive elements of Shapefiles are present. However, it is good practice for Makefile− and Shapefile writers, to make extensive use of the macro facility. Macros provide an extremely flexible yet simple means to represent structural aspects of a complex software system. In our example, there are macros that identify the program, its source objects, objects that are included by others, and corresponding derived objects. The actual dependency lines are just macro applications that are almost generic. This genericity is most important for the development of Makefile and Shapefile *templates* described later in this tutorial. Another big advantage of

*) This default behavior is effective from shapeTools-1.3 onward. Previous versions of shape try to access busy-versions (regular files) and give up if these do not exist.

this kind of information structuring is that structural information can easily be extracted for use by other tools or the development environment. Shape allows to extract this information with its *echo option* which allows to examine any macro:

```
jim> shape -echo SOURCES
tif2ps.c scantif.c genps.c
```

17. Derived Object Management and Cooperative Builds

The typical activity carried out by shape is, to drive the compilation process for the various source objects of a software system in an intelligent manner. Derived objects that result from shape-driven compilations are generally stored in a *derived object cache*, where they are available for possible later retrieval. Shape not only stores the objects that result from compilations, but also all relevant parameters of compilations, such as the compiler that was used and the compile flags that were in effect during the compilation.

Shape's derived object cache is most valuable for projects that employ many programmers who share the same source repository, and thus the same derived object cache. Sharing cached derived objects allows the programmers of a development team to profit from previous compilations initiated by other programmers. If, for example, Sally decided to do the same thing as Jim did in our example above, she'd look at the following build log, produced by shape:

```
sally> shape
... tif2ps.o[702743241.11300.1] restored from cache
... scantif.o[702746105.11550.1] restored from cache
... genps.o[702743262.11300.2] restored from cache
... version.o[702746115.11550.2] restored from cache
... samprog[702746118.11550.3] restored from cache
shape - 'samprog' is up to date
```

In the described example, Sally benefits entirely from the previous build performed by Jim. When shape is used as build-driver, most deriveds are compiled from some immutable source version in the repository. If the same compiler is used with the same switches, the result can be used safely by other programmers. Shape constructs a *derivation key* attribute for each object that results from a compilation. This attribute is supposed to contain all relevant parameters of a derivation. How this attribute looks like is shown in the figure below. In projects where many programmers work together on a large program, consisting of many modules, derived object management is a big time saver — especially if the developers are using networked workstations. While each programmer typically modifies only a small number of source objects, he or she needs to compile the entire program in order to unit-test local modifications (see below). With shape and shared repositories, each build will mainly consist of restoring previously compiled objects.

```
sally> vl -format '$__self$:\n\t$____DerivationKey__$\n' scantif.o
scantif.o[busy]:
    AV(1.0)tif.h[671888449*]\
    defs.h[671888446*]scantif.c[671888449*];\
    @(.o:.c){CC{cc}CFLAGS{-O -DUNIX -DBIG_ENDIAN }};sun4
```

Shape's derivation key attribute* is another example of a *user defined attribute* in the underlying version object repository, AtFS. The attribute in the figure above ("__DerivationKey__") is constructed from the following ingredients:

*) The leading and trailing underscore characters are used in names of shapeTools internal attributes.

- a *magic number*, identifying the syntax version of the derivation key attribute ("AV(1.0)").
- the object identifiers of all objects upon which a derived object depends (*dependency objects*). The object identifier consists of the dependencies' names (e.g. "tif.h"), the numerical representation of the time when the version was last modified (e.g. "671888449", representing Wed Apr 17 13:40:49 1991), and an attribute expansion indicator ("`*`"). If present, the attribute expansion indicator says that attribute expansion was turned on when shape retrieved the source object from the version repository.

  In case the dependency object is itself a derived object (rather than a source object such as "tif.h"), the object's *cache key* attribute will be used instead of the source version number.
- an identifier for the *derivation rule* that was used to compile the derived object. In the example above, the rule identifier is "`@(.o:.c)`", identifying an *implicit target rule* (see section on *Implicit Rules* below) which produces an object with the name extension ".o" from a corresponding source object with the name extension ".c".
- the set of relevant macro definitions affecting the transformation. In the example above, the implicit rule uses the macros "`CC`", and "`CFLAGS`". Which macros are considered "relevant" can be controlled by the user (see section on *default rules* below).
- the hosttype (or target architecture) this derived object was made for ("sun4"). See also the discussion of shape's built in HOSTTYPE macro in the section on *default rules*.
- the type extension for the derived object (".o")
- all relevant macro definitions that were in effect when the compilation took place (in this case, the macros CC, CFLAGS, and vflags were preserved).

Note, that the primary source (in the example "scantif.c") is treated as just another dependency. If any of the stored parameters is changed for another build (let's say that different source version are selected, or CFLAGS is set to define "-DLITTLE_ENDIANS"), shape will try to recover an appropriate derived object from the cache, or compile a new derived object version, and mark it with an appropriate derivation key.

17.1. Derived Object Cache Housekeeping

The *special target* ".BPOOL:" in the Shapefile can be used to control exactly which derived objects shall be cached and which shouldn't. If – for example – a program consists of many very small source objects, it may be appropriate to turn caching off, because the cache lookup overhead exceeds the compile time. This is done by specifying ".BPOOL:" with an empty dependency list. If the program consists of a number of small modules but also has some big source objects, it would be smart to cache only the big items. List the object names to be cached as dependencies to ".BPOOL:". If a system consists entirely of bulky source modules, caching should always be enabled. To do this, either remove the ".BPOOL:" special target from the Shapefile or list all objects as its dependencies.

Once in a while, it may become necessary to clear out the cache entirely. To do this, you shall use the command:

```
$  atfsrepair -C
```

18. Isolating Concurrent Change Activities

Jim and Sally are now ready to apply their changes. For the sake of simplicity, we assume that Sally is in charge of "tif2ps.c", and "scantif.c", while Jim is responsible for a change to "genps.c". Both go ahead by executing the *reserve-modify-build-test-deposit* cycle described in the version control tutorial. Note that Jim, Sally, and samprog_admin shall share the same Unix group-Id, and /usr/projects/samprog/AtFS be group-writable. In this phase of our project, all programmers work concurrently on various changes that shall be individually tested (*unit-test*) together with the original (unmodified) versions of all other system components.

When shape is called within the development workspaces, it builds different ‘‘samprog’’ versions for Jim and Sally: Sally’s version of ‘‘samprog’’ is built from her modified versions of ‘‘tif2ps.c’’, and ‘‘scantif.c’’ (both being respective files in Sally’s development directory), and the original (most-recently saved) version of genps.c, stored in the version archive. Jim’s ‘‘samprog’’ test version is built from the busy version of ‘‘genps.c’’, including his modifications, and the original versions of ‘‘tif2ps.c’’, and ‘‘scantif.c’’ in the archive. As long as Jim and Sally do not save any of their modifications (thereby creating new ‘‘most-recent’’ versions in the repository), their change activities are perfectly isolated. Both can build private versions of ‘‘samprog’’ that include only modifications of the privately modified source objects.

Following this model, however, Jim and Sally have to coordinate themselves in order to proceed from unit test to integration test in an organized manner. While this is generally no unsolvable problem in two-programmer teams, this policy becomes a nuisance (more realistically impossible) for larger projects. However, up to this point, shape hasn’t displayed any of its real capabilities. The Shapefile that is being used is actually a very simple Makefile. As we will see in the subsequent sections, shape offers powerful control over the change process that makes coordination of large programmer teams a reasonable job.

19. Controlling Visibility of Versions

The problem is that shape, by default, binds the most recent version of a source object to the object’s name. If either programmer begins to save intermediate work results, the other will be affected. With the concept of *version status*, the shapeTools system provides a means to assign (development-) process specific meaning to versions of source objects. Versions can be in one of the following states:

- busy (regular Unix file; this is not really a version, because it can be modified without changing its identity. All source objects that undergo immediate change are so called ‘‘busy versions’’ that live as regular Unix files in the workspace of a developer)
- saved (private or intermediate version; purpose: backup for developer; shapeTools-operation: ‘‘save’’)
- proposed (work result, submitted by the developer for approval; shapeTools-operation: ‘‘sbmt’’)
- published (work result, approved for inclusion into the next system version; shapeTools-operation: ‘‘publ’’)
- accessed (version is formally integrated into internal system release, often called prerelease; shapeTools-operation: ‘‘accs’’)
- frozen (version is formally integrated into official system release and probably shipped to customers; shapeTools-operation ‘‘frze’’).

While version status does not impose any restrictions on visibility, it can be used to realize a discretionary hierarchy of scopes. By convention, all versions that have the status ‘‘saved’’ are private affair of the developer who created them. If another developer chooses to ‘‘see’’ private versions, he is free to do so, but expected not to complain if such a version is replaced by a new one that doesn’t work any more. Depending on the size and organizational structure of a project there will be different schemes for controlling who relies on what.

While small projects, such as in our example, will find it unnecessary overhead to have formal QA approvals, medium sized projects will consider this essential. A good rule of thumb for developers is to rely on source versions that have been published, i.e. are dedicated for use by others. It is easy to implement this policy with shape: we introduce a *version selection rule* that discriminates private versions of other developers. A selection rule that does just that is:

```
developing:+
        eq (status, busy);
        gt (status, saved), max (version).
```

Spell: *For all sources, try to bind a busy version. If this is not possible, bind the most recent version with a status better than "saved".*

Such a rule is usually sufficient to isolate change activities of different developers within a project, while they are busy implementing and unit testing their changes. Before we will continue the example with a discussion on how the project proceeds from development and unit testing to system integration, we shall have a closer look at the Shapefile.

20. The Shapefile (and the Makefile)

It has already been implied that shape's behavior is controlled by the Shapefile (just as make is controlled by the Makefile). Shapefiles are the central place where structural information about a managed system is stored (identification of the components and their interdependencies). With respect to this role, we refer to the Shapefile as the "system model". Besides the system model part, Shapefiles can also contain *variant definitions*, and *version selection rules*. There are also some shape specific points about default target rules. (sometimes called *Implicit Rules*). The different elements of Shapefile will be explained in depth in the subsequent sections on *Version Selection*, *Variant Definition*, and *Implicit Rules*.

Shapefiles look largely like Makefiles − shape can even be fed with a standard Makefile. System descriptions, target specifications, dependencies, tool scripts, and regular macro definitions in Shapefiles are just about the same as in Makefiles. In fact, it is a good idea to keep as much system description information as possible in Makefile, and *include* it into the Shapefile which should only contain shape specific descriptions, and – of course – include directives.

While the *syntax* of the control files is very similar, there is an important semantic difference between make and shape: object names appearing in a Makefile directly reference files in the current directory. Shape, in contrast, references *abstract objects*. Abstract objects in the Shapefile must be *bound* to a particular version before any tool can be applied to them. How object names are bound to versions is determined by shape's version selection rules.

21. Version Selection

While it is possible to combine make with one of Unix' standard version control systems (SCCS or RCS), the functionality of these combinations is rather limited. Make can only access the most recent, or a precisely known (i.e. with number) version within RCS or SCCS archives.

Shape, in contrast, can dynamically select versions that meet certain criteria. These criteria can be flexibly specified for each individual component. Before a manufacturing step is executed during a build, shape takes a look at every source component involved, retrieves the right version from the archive, and installs it in the build directory. When a tool (e.g. a compiler) is activated, it will find the right version. When the manufacturing step terminates, shape makes sure that the original state of the build directory is restored (files that would have caused name conflicts are temporarily moved).

Within the Shapefile, version selection rules are defined similar to target rules, but with the “:-” character sequence as rule definition symbol.

```
most_recent:-
      eq (status, busy),
         msg(Using busy version of $+);
      max (version),
         msg(Using last saved version of $+).
```

Criteria for version selections are specified in a simple rule language. Each rule consists of a name (“most_recent” in the example above) and a body. Rule Name and -body are separated by a colon followed by a minus. The rule body is made up by a sequence of *alternatives* that are separated by semicolons. The rule body is finished by a period. Alternatives consist of a comma separated list of simple *predicates* that express basic constraints in terms of *attribute expressions**. A formal definition of the selection rule syntax is given in Appendix B.

21.1. The Process of Version Selection

Shape performs version selection object by object, when a dependency line is evaluated. Assume the following dependency line from a Shapefile:

```
samprog: most_recent tif2ps.o genps.o scantif.o
    $(CC) -o samprog tif2ps.o genps.o scantif.o
```

When the target “samprog” is requested (by issuing the command “shape samprog”), shape analyzes samprog’s dependencies. The first item in the dependencies list, “most_recent”, is the name of a selection rule. This causes shape to set “most_recent” as the *current selection rule*. The following dependencies − tif2ps.o, genps.o, and scantif.o − are references to derived objects that are compiled from respective C sources (“tif2ps.c”, “genps.c”, and “scantif.c”). Once that shape has tracked down the abstract source objects for a requested compilation, they will be bound by the current selection rule.

If shape, on its recursive descent through a system’s dependency hierarchy, encounters a dependency line with another selection rule activation, this rule will be set current only for the subsequent productions. Once the target that activated the new rule has been made, the previous selection rule (hierarchically superior) will be restored as current rule. Experience shows, however, that one rule activation is usually sufficient for an entire shape run. Thus, it makes sense to use shape’s *−R <rule name>* option to define the selection rule to be used. Specifying the selection rule from the command line is not only more convenient, but also provides more flexibility in controlling the configuration identification process.

---

*)The selection rule machinery is based on the concept of *attributed software objects* (ASO) that is implemented by AtFS, the underlying version object repository. It is important to keep in mind that all objects (files and versions) are ASOs and thus have a number of *attributes*. Some of these attributes are built into the system (e.g. name, version, status), but arbitrary other other attributes can be defined by the user. A table with all standard attributes of the shapeTools-1.4 system is given in Appendix A.

21.2. Evaluation of Shape's Selection Rules

The binding process proceeds as follows: Each dependency name is resolved to a corresponding source object name (e.g. an object name ''foo.o'' could be resolved to ''foo.c''; see section on Implicit Rules). The source name is passed to the selection machinery, and sequentially matched against the patterns of the different alternatives. When a pattern* matches the name, shape retrieves all versions of the object into an internal set (*hit set*). The subsequent attribute predicates are *discrimination operators* that constrain the set of versions. All objects that do not meet the criteria expressed in a predicate, are removed from the hit set. Thus, the sequence of predicates in an alternative forms a logical AND expression of constraints over the ASOs in the hit set. While most predicates constrain the actual hit set, it is also possible to enforce *external constraints* that must also be complied with, in order to have a valid selection. The predicates that impose external constraints are *exists*, *existsnot*, and *existsuniq*. These predicates aren't evaluated against the current hit set, but have hit sets of their own. If an external constraint is fulfilled, the current hit set is left unchanged. If the external constraint fails, the current hit set will be set empty.

An alternative can *fail* or *succeed*. An alternative succeeds, if the hit set contains exactly one object that satisfies all constraints expressed by the predicates. If no such object can be found (hitset empty), the object set is insufficient (e.g. in case of a system release not all current versions may have passed QS). If the set contains more than one object, our version binding is ambiguous. If a rule alternative does not succeed, it fails, and the next alternative is tried. An alternative fails as soon as the hit set becomes empty. Predicates that appear after the set is empty, will not be evaluated. If all alternatives have been tried without success, shape reports a configuration error and aborts. The working of selection rules is described in detail in the manual page **bindrules(7)**.

Now that the functioning of shape's selection rule machinery has been explained, let's go back to the example project, and see how the selection rules are used to drive somewhat less trivial development tasks.

22. Proceeding from Unit Test to Integration Test

While simple unit-testing is − from an organizational point of view − straightforward, integration testing is far more complicated. The process of system integration typically depends on the particular organization of the project, involves policy issues (such as different responsibilities for testing, QS, change approval etc.), and reflects administrative, and management competences. The scenario described in this tutorial is intended to illustrate some of shape's capabilities to aid these processes. The particular setup is necessarily simplified and shows only one (of possibly many) simple example of how the involved problems can be approached with the shapeTools system.

Before we continue our tutorial example, we shall briefly outline the general idea of integration testing as assumed in this example. Conceptually, the starting point for integration testing is the previous version of the system, called the ''home-baseline''. Changes are now incrementally integrated into this baseline, one modified module at a time. So, at any given point in time (during our integration phase) we have some intermediate system version that contains some, but not all changes that have been made. We call this the ''baseline-of-scrimmage''. If the various test runs against the current scrimmage-baseline succeed, the next change is integrated a.s.o. until all changes have been incorporated and successfully tested. This is what we call the ''goal-baseline''*. (For the sake of simplicity, we have omitted the ''unlikely case'' that a certain modification fails a test, ;−) Technically, the incremental integration of modifications into the

---

)The first predicate of an alternative may be a pattern, similar as in ed(1). Specifying no pattern is equivalent with a pattern that matches everything ('''').

*) The football-ish terminology is in no way standardized. Its only purpose is to illustrate the process of

scrimmage-baseline is implemented as numerous builds of the system to be tested (“spins”). While shape is not directly a testing tool, it definitely makes this incremental build process easy.

After Jim and Sally have reached a point in their work when they believe that the required changes have been successfully implemented (passed unit test), they *submit* the corresponding versions of their sources for integration. Submitting a version is done by executing the “sbmt” command of the shapeTools version control system on the respective version of the source module.

```
jim> sbmt tif2ps.c
```

Note, that Jim and Sally produced quite a number of intermediate, private versions of their modules during development and unit-test. These intermediate versions can usually be thrown away to avoid unnecessary cluttering of version control reports (“vl”). Typically, only the latest of these private versions needs to be retained for the purpose of integrating it into the next system version.

22.1. Effective Use of Version Status

The effect of submitting a version is nothing but a *change of status* from “saved” to “proposed”. The version status is one of the built-in attributes of the shapeTools system.

We will use the status attribute to drive controlled builds of “samprog” that successively integrate the versions that have been submitted by our developers. For this purpose, we extend the sample Shapefile, above, by a *version selection rule* that *binds* the latest versions with status “proposed” to the source objects’ names. For source objects that haven’t been modified, we want to select the version from the home-baseline.

One problem we’re facing is that possibly all of our developers have submitted versions of their modules for integration testing. So, selecting all “proposed” versions for a build would result in a *big bang* integration, making it very hard to trace defects to particular modifications. In order to integrate modules one by one per build, we need an additional *user defined attribute* to mark the particular version that is going to be integrated in the next spin. We will call this attribute the “test_switch”. If it is switched “on”, the particular version will be used for the build. The attribute is set with shapeTools’ “vadm” command:

```
jim> vadm -attr test_switch=on genps.c[1.5]
```

The recipe for the next baseline-of-scrimmage build is:
for all source objects
– *bind all new versions that have already passed the test*
– *bind the version(s) that shall be integration-tested in this spin*
– *bind all remaining objects to the version from the home-baseline.*

A shape version selection rule that implements this recipe is shown in the following figure:

integration testing that often requires just as much guts as football playing.

```
i_test_rule:-
      gt (status, saved), max (version),
      eq (status, published);
      eq (status, proposed), eq (test_switch, on);
      ge (status, published), max (version);
      cut (Cannot bind $+ -- something's wrong here!).
```

The first alternative in this rule is the most interesting one: its purpose is to bind all new versions that have already passed the test. To do just that, all versions that are proposed or better are retrieved from the repository. From this set, the alternative singles out the most recent version. If the integration test for this version is completed, it should – according to our policy – have the status *published*. This alternative succeeds for published versions of the previous release if no new changes have been proposed for the module in question, or if new changes – resulting in a version – have been approved (i.e. published). It will fail otherwise.

Our policy shall be that once a new version has passed the integration test suite, it will be set to status ‘‘published’’, and the test_switch attribute will be removed:

```
jim> publ genps.c[1.5]
jim> vadm -delattr test_switch genps.c[1.5]
```

Removal of the test_switch is important because the second alternative would fail if more than one proposed version has this attribute set to *on*. As an exercise, define a somewhat more ‘‘foolproof’’ version of alternative two.

If the procedure is clear and becomes routine, it is easy to automate. In fact, it is strongly recommended to automate procedures such as above, in order to ensure consistent application. Later we will provide an example, of how shape can be used to automate large parts of, or even the entire procedure.

Remember that this policy is only one of many possible ways to run an integration testing process. Definition of policies, such as which attributes are set, when, and by whom is strongly influenced by the needs of a particular project and thus needs to be done in different, appropriate ways.

23. Managing System Variants

Although there is no precise, common definition of the notion *variant*, everyone who has ever participated in the development of a ‘‘real program’’ knows the problems that come up when a product spawns specialized, concurrent versions. For this discussion, we will assume that a variant is an entity that serves a certain, invariant purpose with respect to varying sets of constraints. An invariant purpose in this sense could refer to a module interface that is implemented by different algorithms. It could as well be a complex service, such as a word processing system that specializes for different languages, or comes with different functional bandwidth. In practice, this usually translates into the concurrent* existence of a *standard version*, and a number of somewhat modified versions, aimed at different markets or embodying specific enhancements/adaptations for assorted customers. In the following discussion we will refer to these versions as *variants*. Note that the term *variant* can be misleading, as it generally refers to specific variations of a function or service (implemented by a software system) that is actually *invariant*, i.e. the common concept present in all variants. Another thing worth keeping in mind is that variants occur on component and aggregate levels: a composite system may consist of the same components that are variant (component variant), or it maybe composed of different components which themselves need not be variant (aggregate variants). Keeping system variants under control is often a difficult task. Even more difficult is the problem of *change management* for variants. While certain changes are only applicable to one variant, other changes may affect some or even all of them. If product variants are organized as complete, redundant source trees (i.e. components that are common to some or all variants, are copied), the *multiple maintenance problem* is invoked. Propagating changes to common (*invariant*) system parts of a multi-variant product structure is a notoriously tedious and error prone task. Automatic or semi-automatic change propagation can help but introduces other problems (keeping track of the propagation, eventual corruption of the source, etc.). It is much more efficient to organize system variants in a redundancy free manner. In the following sections we will describe, how ShapeTools can help to tackle these problems by providing the capability to build many different variants of a software system from a shared, low-redundancy product repository.

23.1. Organizing Variants

Technically, variations of a software system are commonly represented by

– different system compositions
– variant source separation (‘‘*variant segregation*’’)
– single source variants
– derivation variants
– combinations of the above.

In the case of *different system compositions*, certain components of a system may be replaced by others, some components may be present that are otherwise not part of the system, while other components may be omitted. The figure below shows three different variants of a system (S_i) that are composed differently.

S_0 = { Ma, Mb_0, Mc_0, Md }
S_1 = { Ma, Mb_1, Mc_2, Md }
S_2 = { Ma, Md, Me }

Aggregate variation occurs only on the composite level of a system. It refers to the structural effect of ‘‘recipe variations’’ for system builds. The *replacement* of particular components by component variants (example S_0/S_1) is referred to as component variation (in contrast to aggregate variation). Component variation has to be further differentiated according to two commonly used representation techniques: *variant segregation*, and *single source variants*.

With variant segregation, each variant of a component is maintained as a physically separate copy. Variant segregation introduces a certain amount of redundancy into the product structure, because a variant is often a modified copy of some other source object. As these modifications are often small compared to the amount of data that the variants have in common, this technique is vulnerable to the multiple maintenance problem. Modifications that apply to common parts must be propagated to the other affected variants.

In the case of *single source variants*, only selected parts of a given component are included in one variant, while other parts are included in another variant. This technique is often implemented by preprocessing a given source component. In a number of programming languages, such as C, this technique is applied as *conditional compilation*. While this variant technique has the big advantage of avoiding redundancy, it often introduces the problem of cluttered sources.

---

*) i.e. without one variant being the successor or predecessor of another.

With *derivation variants*, the same source component is processed by different compilers in order to adapt to different computing platforms. Derivation variants often occur implicitly if a system is built within a heterogeneous computer network (this is, for example, a common problem with Sun-3 and Sun-4 computers in a common NFS file system). Explicit derivation variants occur typically in cross compilation scenarios.

This tutorial is not intended as a cookbook for dealing with variants. Which techniques for variant handling are to be applied when and where (e.g. using conditional compiles in the same source unit vs. physical separation of variant components) must be decided in an appropriate way for each project. It is always a good idea to *plan* these decisions in advance, or to create guidelines how to handle these problems. However, especially variants have the tendency to occur suddenly, and at places where they weren't meant to be. All too often, the arising problems must be dealt with in an ad hoc fashion. ShapeTools may offer good help here.

23.2. Shape Variant Definitions

Shape allows to handle the above mentioned variant management techniques in a simple and efficient manner. The key is shape's powerful *dynamic macro re-definition facility* together with a highly parametric tool control process. In Shapefiles, macros are used to define just about anything, such as:

– the list of a system's source components
– the list of the corresponding derived objects
– the tools that perform the derivations
– the flags and command line switches that control the tools, as well as
– the search strategy used by shape to retrieve the referenced objects from the repository.

Dynamic control over the definitions for all these macros allows to implement the techniques described above effectively. Because of its close relation to the purpose of variant handling, the respective construct in the Shapefile is called *variant definition*. Shape's variant definitions allow, however, to generally manipulate macro definitions depending on the flow of control during a build. This is in no way restricted to variant administration. A particular advantage of variant definitions is that variants appear as logical entities (represented by variant names) that abstract away from the particular technique(s) that are used to implement them. Moreover, shape provides a very flexible way for *combining* variant properties of a system, while incompatible combinations can be prohibited.

The figure below shows three simple variant definitions:

```
standard:+
      vpath= standard
      VARSOURCE= seturg.c bootstuff.c speedboat.c mortf.c
      VARINCLUDE= fis.h dparam.h
      DEPENDENCIES= standard/Dependencies
      VARCFLAGS= -DSTDAPP

  smith:+
      vpath= smith
      VARSOURCE= smithboot.c storeif.c creport.c
      VARINCLUDE= cformats.h
      DEPENDENCIES= smith/Dependencies
      VARCFLAGS= -DSMITH
      COMPILER= prop

  miller:+
      vpath= miller
```

Variant definitions in a Shapefile are simply groups of macro definitions that become effective when the respective variant is activated.

24. Activation of Variants

There are two possible ways to activate variants for a build run: they can be requested from the command line with the **−V** <*variant name*> option, or be attached to certain targets. In the first case, the variant activation is *global* for the entire build process. In the latter case, variants apply only to the derivation process of a target. Target specific variants are activated in a similar way as shape's version selection rules: they are simply put on the target's dependency line. Other than selection rule activations, however, variant activation dependencies are marked with a "+" sign as prefix to the variant name.

```
samprog: +standard $(OBJECTS)
smiths_samprog: +smith $(OBJECTS)
millers_samprog: +miller $(OBJECTS)
```

If any of the targets in the example above are requested, shape will find a variant activation as first dependency of a target, and thus define (or redefine) the macros that belong to the variant. Macros that are defined by a variant activation can be referenced just as any macro in shape (or make). The definition of such a macro is valid for all subsequent productions. When the target that activated a variant is completed, the respective macro definitions are set to the previous state.

When a variant is activated, the respective macro definitions *replace* possible previous definitions. I.e. if a macro *SAMPLE* is defined as

```
SAMPLE = foo
```

on the top level of the description file, but is also defined in a variant as

```
SAMPLE = bar
```

the macro will be substituted as ‘‘bar’’, as soon as the variant is activated.

Because variants occur in many interesting ways at the same time, shape allows to have *multiple active variants*. Consider the case that there are several customer specific variants of a system, let’s say *standard*, *smith*, and *miller*. While these variants are certainly mutually exclusive, there may be other variant classes that could for example be used to control the quality of the code:

```
# available compiler systems

gnu:+
    CC = gcc -Wall
    OPTIMIZE = -O2 -inline-functions
    DEBUG = -g -g3
    PROFILE = -pg -a
    STDC = -ansi

prop:+
    CC = cc
    OPTIMIZE = +O3
    DEBUG = -g -z +Y
    PROFILE = -G
    STDC = -Aa

# code quality options - may depend on selected compiler

debug:+
    VARCFLAGS = $(DEBUG)

profile:+
    VARCFLAGS = $(PROFILE)

optimize:+
    VARCFLAGS = $(OPTIMIZE)


malloc_debug:+
    VARINCLUDE = -I/usr/local/debug_include
    DEBUG = -DDEBUG_MALLOC
    VAROBJECTS = /usr/lib/debug/malloc.o
    VARLIBS = /usr/local/lib/libdbmalloc.a

CFLAGS += $(VARCFLAGS)
```

During development, it does make sense to combine different of the variants above. The next figure shows a somewhat more generic Shapefile target line that allows to activate appropriate variants that are themselves defined in macros. These macros may be set from the command line – or even from other variant activations:

```
samprog: +$(CLIENT) +$(COMPILER) +$(QUALITY) +$(STATS) $(OBJECTS)
```

A typical shape invocation that takes advantage of this construction would look like this:

```
sally> shape CLIENT=smith QUALITY=debug STATS=profile
```

A more convenient way to request the same variants from the command line would be, however:

```
sally> shape –V smith –V debug –V profile
```

As the latter form of variant activation provides more flexibility (and is more convenient) it should be preferred in daily use. The first way of variant activation is more useful for those complicated cases, when target specific variant activations are needed, depending on the activation of other variants.

In the example, the variants *smith*, and – indirectly – the variants *debug*, and *profile* define the macro *VARCFLAGS*. However, in this case the macro definitions are not replaced, as would have happened with regular macro definitions, but *augmented*. With all three variants active, *VARCFLAGS* would – for example – be substituted as follows:

```
$(VAR_CFLAGS) → –DSMITH –g –z +Y –G
```

It is obvious that some care should be taken to avoid variant activation profiles that are contradictory, such as activating variants for different customers, or for multiple computing platforms at the same time. Shape by itself has no means to understand which switch settings can be meaningfully combined, and which can't. In order to provide plausibility checking of variant combinations, shape allows to specify *variant-classes*. Variant classes are groups of mutually exclusive variant definitions. In completion of the current example, the following variant classes would be appropriate:

```
vclass customer ::= ( standard, smith, miller )
vclass compiler ::= ( gnu, prop )
vclass quality  ::= ( malloc_debug, optimize )
vclass stats    ::= ( profile, optimize )
. . .
```

If an attempt is made to combine any two variants that are member of the same variant class, shape will complain about it.

So far it has been shown, how certain macro definitions and redefinitions are made effective, and how they can be combined. In order to take ultimate advantage of dynamic macro (re–) definition, it is useful to employ a small number of techniques and conventions. In the next few sections it will be described and demonstrated how these techniques and conventions can be applied to control the different variant administration techniques mentioned earlier.

25. Implementation of Aggregate Variants

If the composition of a software system is described by component lists – represented as shape macros – it is easy to manage aggregate variants. While the particular organization of the various parts of a system's composition list may be different for each project, the following Shapefile macros may exemplify a composition list setup for projects that use the C programming language:

Macroname	Purpose
SOURCES	List of all directly compilable C source files
HEADERS	List of all include files that are needed to compile *SOURCES*
OBJECTS	List of all compiled, loadable modules needed to build a program (or library)
LIBS	List of additional libraries needed to build a program

These macros shall be defined in the regular section of the Shapefile (or a Makefile that will be included by shape) to contain only those components that are common to all variants. The portion of the system model that describes the parts common to all variants could look like this:

```
SOURCES = tif2ps.c scantif.c genps.c
OBJECTS = tif2ps.o scantif.o genps.o
INCLUDE = tif.h defs.h
LIBS = /home/development/basis/geomodel.a

samprog: $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LIBS)
```

So far, the system model has nothing shape-specific in it. In fact, it could as well be processed by make. Thus, it is a sound idea to place basic structural data in a control file called *Makefile*. A *Shapefile* which would introduce the shape specific controls, and definitions would look like this:

```
include $(BASE)/shapelib/selection_rules.shp
include $(BASE)/shapelib/variant_defs.shp

include Makefile

SOURCES += $(VARSOURCE)
OBJECTS  = $(SOURCES:.c=.o) $(VAROBJECTS)
INCLUDE += $(VARINCLUDE)
LIBS    += $(VARLIBS)

include $(DEPENDENCIES)
```

If the target ‘‘samprog’’ is requested with the command:

```
sally> shape samprog -V smith -V debug -V malloc_debug
```

shape will activate three variants, and thus dynamically define the macros *VARSOURCE*, *VARINCLUDE*, *VAROBJECTS*, and *VARLIBS*. If any of the macros is undefined, shape simply ignores them. Assumed, the customer variant has been set to ‘‘smith’’ and the quality variant to ‘‘debug’’, and ‘‘malloc_debug’’, the *OBJECTS* macro would substitute

```
$(OBJECTS) → tif2ps.o scantif.o genps.o \
  smithboot.o storeif.o creport.o /usr/lib/debug/malloc.o \
  /usr/lib/debug/mallocmap.o
```

Note that the *OBJECTS* macro substitution will be computed from the string substitution of the *VARSOURCE* macro, defined in the *smith* variant, and the *VAROBJECTS* definition of the *malloc_debug* variant.

25.1. Keeping Variant Specific Components in Separate Directories

Shape provides a feature that is particularly helpful to avoid confusion of the source tree when structurally disparate variants of a software product have to be maintained, the *variant search path*. The key idea is simple: maintain all components that are common to all variants in the system's main source directory; for each aggregate variant, create a separate subdirectory (within the main directory) with the name of the variant (e.g. ‘‘smith’’, or ‘‘miller’’), and keep those special components there (see figure below).

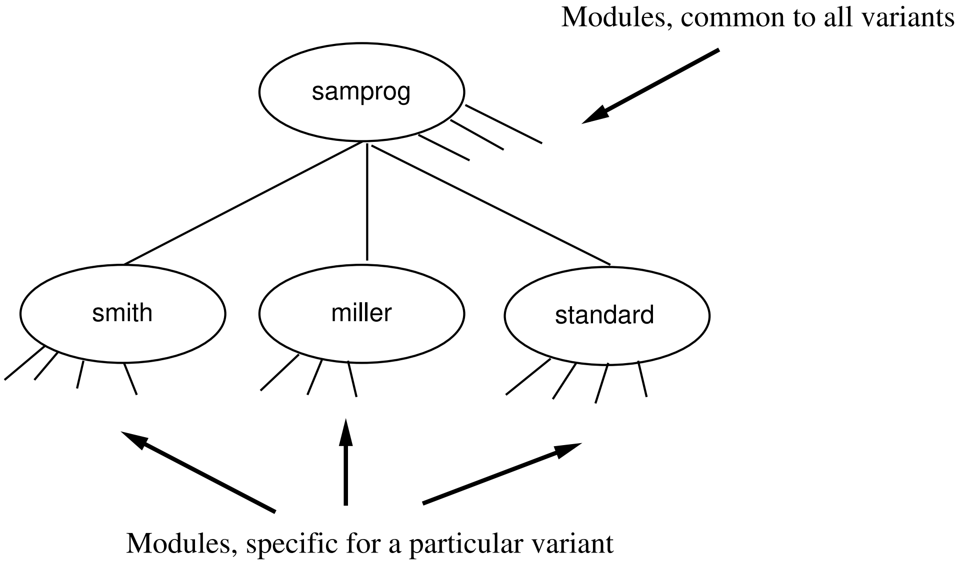

Fig.7: Variant Components in Separate Subdirectories

The variant subdirectories are the place where the version archive directories should be maintained. The development process for variant components stored in these directories is just as any normal development. The components are treated as separate objects with their own private version history. System builds, however, shall be run from the main directory. Shape uses the special macro *vpath* (lower-case!) to extend its vision of components. The *vpath* is defined in the variant definition (see example above).

25.2. Vpath Search Strategy

If the vpath is non-empty, shape tries to find any referenced source object *in the vpath directories first*. If several variants are set active that define vpath, the vpath is concatenated and searched from the right to the left, i.e. the last variant that has been activated has precedence. Only if a referenced source component cannot be found in any of the vpath directories, the current (main) directory is searched. If a source object has been found, it will be bound by the current version selection rule, and be temporarily *installed in the build directory*. This means that components which are maintained in a vpath subdirectory are temporarily moved up to the main directory. Thus, it is not necessary to make any reference to a vpath subdirectory path in the dependency specifications.

As suggested in the example above, it is advised to keep separate files that describe the variant specific component dependencies. These files should also be stored in appropriate *vpath* directories. This is necessary, because systems with variant composition lists will also have a variant dependency structure between the components.

26. Implementation of Single Source Variants

Single source variants are implemented by preprocessing the same source object contents in various ways. A well known example of this variant management technique is *conditional compilation* as frequently used for C programs. In order to make this technique work effectively, the source text has to be prepared carefully by inserting preprocessor instructions (e.g. C's `#ifdef` directives) at appropriate places. The source text that will actually be compiled is determined by the instructions given to the preprocessor tool.

Shape assumes that preprocessor instructions are passed as command line options to the preprocessing tool. Variant control is again based on the idea of keeping these variant specific tool options in special macros that are substituted within the production scripts. In the example above, the macro *VARCFLAGS* is used to store special options for C compilations that apply for a certain variant. If, for example, shape is requested to build the smith variant of ''samprog'', `VARCFLAGS` is defined as `−DSMITH`. It can now be substituted in the production script controlling the tool invocations. A typical production script that drives C program translations is shown in the example below:

```
$(CC) −c $(CFLAGS) $*
```

Note that the production script is completely generic, i.e. neither the compiler, nor the compile options or the source to be compiled are stated explicitly. By convention, shape - as make - uses the macro `CFLAGS` to pass options to the C compiler (see also section on *Implicit Rules*, below).

There is no predefined convention with respect to macros used to pass variant specific options. In many cases it will be straightforward to simply redefine the `CFLAGS` macro (or any of its programming language specific counterparts) within the appropriate variant definitions. However, care should be taken not to loose any previous definitions of `CFLAGS`, made outside of the variant definitions. A more sound solution to the problem of variant option passing is the use of shape's *cumulative macro definitions*:

```
CFLAGS = −myflag −s
CFLAGS += $(VARCFLAGS)
```

In this example there is a regular (non variant) definition of the `CFLAGS` macro, augmented by the contents of the variant options macro `VARCFLAGS`. The cumulative macro definition operator is by no means restricted to use with variant macros. It simply *concatenates* the string on the right of the operator to the current definition of the macro, rather than replacing the previous definition by a new one.

While the control of conditional compilation described in the above examples is a bit biased towards the use of the C programming language, it can be generalized to many other programming languages as well. Although not all programming languages offer sophisticated preprocessing as ''part of the language'' it can easily be added to any compilation process (see section on *Implicit Rules*).

In larger project settings it is usually worthwhile to spend some effort to plan for variants in advance, and to provide adequate support for techniques to handle them. As far as component variants are concerned, appropriate compile scripts (and corresponding coding conventions) should be developed that take advantage of preprocessing facilities.

27. Implementation of Derivation Variants

The concept of *derivation variants* is related to that of preprocessed sigle source variants. In contrast to this type of source variants, however, derivation variants need no special preparation of the source code but rather employ different tools for obtaining derivatives. From the technical viewpoint, shape implements this sort of variant just as it implements preprocessed variants: by redefining certain (standard-) macros identifying the tools used for the compilation process.

The following example shows three different derivation variants:

```
#
# vclass compiler
#

vclass compiler ::= (pcc, gnu, gnu_ansi)
pcc:+
    CC=cc
gnu:+
    CC=gcc -traditional -W
gnu_ansi:+
    CC=gcc -ansi -Wall
```

The variation of the compilation process is simply implemented by redefining the *CC* macro which – by convention – defines the name of the compiler used for compiling C programs (see next section). Note the initial *vclass* specification indicating that the three variants are mutually exclusive. Activation of the desired derivation variant happens in exactly the same way as described above.

The concept of derivation variants is useful for keeping track of tool versions (at least to a limited degree), and for controlling cross compilation processes. Shape's derived object management is capable of keeping control over which derived objects have been produced with which tools, and under what premises.

28. Implicit Rules

So far, this tutorial described shape's basic syntactical elements: the version selection rules, the macro mechanism, the variant definition facility, and – en passant – the dependency lines. At first glance it may appear as writing a Shapefile is a complicated endeavor. This is not so. Most rules and definitions are typically wrote once and for all by an expert (probably YOU), and (re-)used many times by the users (the programmers) without ever hurting the users' eyes. This is especially true for the specification of shape's *implicit dependencies* and *implicit rules*. For the programmer, who most of the times copies and customizes some furnished Shapefile *template*, shape's operation will appear to be guided by ‘‘magic’’.

Implicit dependencies are the most important source of shape's knowledge about how which products are obtained from what prerequisites. Because the basic recipe for a class of derived objects is the same, it only needs to be specified once. In the case of C program development there is always a source file that corresponds directly to an object file which is compiled from that source. The correspondence between source and derived (sometimes called *target*) object is established by the convention that the source file has a name ending with the suffix ‘‘.c’’ and the object file having the same name with the suffix ‘‘.o’’.

Shape's implicit dependency specification for C program objects is shown in the figure below:

```
%.o : %.c : +(CC) +(CFLAGS)
     $(CC) -c $(CFLAGS) %.c
```

An implicit dependency consists of three elements, separated by a colon:

– the *target patterns* (“%.o”)
– the *source name template* (“%.c”), and
– the optional *ingredient macro definitions* (“+(CC) +(CFLAGS)”).

The “%” character serves as a dynamic pattern symbol, used as a wildcard symbol in the target pattern. In the source name template and everywhere else in the implicit rule, the “%” character is substituted by whatever character sequence was matched by the wildcard symbol in the target pattern. Specifications of implicit dependencies are followed by one or more lines of *tool script*. Tool scripts define the actual sequence of tool invocations necessary to produce the target object(s) specified in the implicit dependency. In the example above, the tool script is very simple: it consists of one line applying the *CC* and *CFLAGS* macros.

28.1. The Transformation Process

When a implicit rule *fires*, shape substitutes all macros with their currently active definitions, and passes the result to the UNIX shell*. Every single line of a tool script is invoked separately. If multiple lines shall be interpreted by a single shell process, they must be terminated by a backslash character (“\”) to indicate line continuation. Shape assumes that all specified target objects are produced when the tool script of a implicit rule is executed and the return codes don’t indicate any problems (i.e. all return codes are zero).

Implicit rules have a dramatic effect on the effort necessary to specify how a complex system shall be build. For directly derived objects, it is usually sufficient just to rely on them. The Shapefile in the figure below contains all the information necessary to build a program:

```
OBJECTS = tif2ps.o scantif.o genps.o

samprog: $(OBJECTS)
$(CC) -o samprog $(OBJECTS)
```

Note that no explicit reference to any source object is needed. Shape determines the source objects’ names by evaluating the appropriate implicit rule.

28.2. Evaluation of Implicit Rules

When encountering a request for an object (either as explicit target specified on the command line or implicit during the course of a build) shape tries to find it in the version object repository right away, or searches for a recipe (an explicit dependency or a transformation rule) describing how to produce the requested object. In the example above, there is no further explicit description of how to produce – let’s say – tif2ps.o. Before giving up, shape matches the name of the requested object against the target patterns of the implicit dependency specifications, it knows about*. In our example, shape matches the name *tif2ps.o* against the target pattern “%.o” of the implicit rule. After match and substitution, the implicit

*) actually, shape passes each command line to the command processor defined by the macro “`SHELL`”. If the macro is not explicitly defined, shape substitutes “/bin/sh” as default.

*)Most commonly used implicit rules are built into shape, so they need not to be stated in the Shapefile.

rule has the effect as if the following explicit dependency appeared in the Shapefile:

```
tif2ps.o : tif2ps.c : +(CC) +(CFLAGS)
$(CC) -c $(CFLAGS) tif2ps.c
```

Shape now has an idea about how *tif2ps.o* may be obtained. If there is no further dependency of *tif2ps.c* (stated either explicitly or by another implicit dependency, perhaps on *tif2ps.y*), shape treats the dependency – *tif2ps.c* – as source, and tries to *bind the name to a version* in the object base (see section *Version Selection*, above). An implicit rule is considered applicable if the implied sources object(s) exist in the version object repository. If this is not the case, shape continues to match the requested name (*tif2ps.o*) against the target pattern parts of the remaining implicit dependencies. If no applicable implicit rule can be found, shape will – finally – give up and complain:

```
shape - don't know how to shape - tif2ps.o
```

28.3. Implicit Rules and Derived Object Management

Shape's derived object management has been explained in the above section on *Derived Object Management and Cooperative Builds*. Target rules – implicit and explicit – are used to control the behavior of shape's derived object management. A dependency line basically specifies a number of derived objects – on the left side –, and prerequisites plus ingredients – on the right side. As pointed out above, shape stores derived objects in the *derived object cache* for possible later reuse. When a derived object is cached, shape associates a *derivation key* with it, (supposedly) precisely describing the prerequisites (e.g. particular source object versions) and ingredients (relevant macro definitions) that determine the outcome of a compilation.

Once shape made up it's mind that a certain implicit rule is applicable, it re-computes the derivation key that *would be* associated with the derived object if it were produced. Shape then searches the derived object cache with the constructed derivation key. If an object with that key can be found, it is simply restored – rather than expensively (re-)derived. Note, that the (substituted) implicit dependency specification provides all the information needed by shape to compute the derivation attribute:

– the name of the source object (which shape binds to a version)

– the definitions of the macros that are relevant for the derivation.

28.4. Ingredient Macros

As shown in the examples above, large parts of the tool scripts that are passed to the shell are constructed by substituting a number of conventionally defined macros. Since the early days of the Make program, the names for most of these macros are de facto standardized. This is very convenient, because simply altering certain macro definitions allows to change the behavior of Make's standard tool scripts. If, for example, one wants to use a different compiler for C translations rather than the system's compiler, one simply redefines the *CC* macro – let's say to "/usr/gnu/bin/gcc". Another typical use of this feature is switching back and forth between debugging and optimizing compilations by altering the *CFLAGS* macro*.

It is obvious that the maipulation of certain key macros has profound influence on the characteristics and the outcome of transformations. Traditional Make implementations have a severe problem here, because they are unable to keep track of macro changes. So, if the programmer decides to switch to a different C

*)Recall that important parts of shape's variant handling mechanism are based on an extension and automation of this technique.

compiler, or to proceed from debugging to production developement (turning on a completely different set of compile switches) but doesn't touch any of the source objects, Make won't notice that any relevant changes have occurred and declare the previous build to be "up to date". The programmer must take action to convince Make that things have to be done.

Shape, in contrast, is able to keep track of the macro definitions that affect a transformation. So, if the definition of a macro changes – due to manual redefinition or a different variant selection – shape detects that derived objects produced by a certain implicit rule are obsolete.

All of shape's built in implicit rules have the set of relevant *ingredient macros* for a tool script defined. As a rule of thumb, ingredient macros that should be listed in an implicit dependency specification will – most of the times – be the macros that appear in the tool script. However, shape gives a lot of flexibility in controlling which macros shall be considered relevant for a transformation. Let's have another look at an example:

```
samprog: $(OBJECTS) $(LIBS) : +(CC) +(LDFLAGS)
$(CC) -o $@ $(OBJECTS) $(LDFLAGS) $(LIBS)
```

The example shows an (almost generic) explicit dependency specification taking advantage of shape's ingredient macro facility. Ingredient macros are referenced with a special syntax: the plus symbol in front of the parenthesized macronames tells shape that the entire definition (macro name and current substitution) shall be used to construct the *derivation attribute* for the derived object produced by this rule (*samprog*). Note that the macros *OBJECTS*, and *LIBS* need not be mentioned in the ingredients part because both refer directly to prerequisite objects whose bound identities will be stored in the derivation attribute anyway.

When using the ingredient macro facility, the user should be careful to only consider macros that *really* affect a transformation in order to avoid unnecessary recompilations.

28.5. Complex Implicit Rules

implicit rules are the place where the characteristics of elementary compilation processes are stored. Shape has a lot of built in knowledge regarding how to do "standard" compilations, such as C, Fortran, Assembler, Modula-2, and a couple of other programming languages. However, in projects that use different programming languages, or use other than the standard compilers that require different compile processes, it is often necessary to define special implicit rules.

While shape's built in rules are all rather simple, project specific rules can become quite complex: they may require multi step compilations (e.g. if database schemas compile into header files and access components for a database), or multiple targets may be produced from one or more source objects in a single transformation. Shape's implicit rules allow to specify rather complex compilation processes. As an example for a more complex implicit rule, let's consider a rule for generating parsers from Yacc grammars.

```
%.o %.h : %.y : +(YACC) +(YFLAGS) +(CC) +(CFLAGS)
    $(YACC) -d $(YFLAGS) %.y
    $(CC) -c $(CFLAGS) y.tab.c
    rm y.tab.c
    mv y.tab.o %.o
    if [ -r %.h ] && diff %.h y.tab.h > /dev/null \
    then \
    rm y.tab.h \
    else \
    mv y.tab.h %.h \
    fi
```

Assume, there exists a yacc grammar source object *parse.y*, and during a build, either a corresponding *parse.o* or the token definition header file *parse.h* are requested. In either case, shape finds the example rule applicable. When the tool script is run, both derived objects – *parse.o* and *parse.h* – are recreated and stored in the derived object cache. Shape only knows about specified objects – objects that are created intermediately during the execution of the tool script are of no concern to shape. These intermediate objects (*y.tab.c, y.tab.o* and *y.tab.h*) are not stored in the derived object cache.

The derivation rule above contains a subtle optimization that could suggest that shape had a problem: the specified derived header file containing the token definitions is not always recreated, because the set of known token classes typically changes far less frequently than the structure of the input grammar. To avoid excess recompilations of components depending on the token definitions, the header file – although actually always rederived together with the parser code – is only *recreated* if it actually changed. In order to avoid caching of the – unchanged – but supposedly (according to the transformation rule's signature) rederived header file, shape checks the time when the implicit rule's toolscript was invoked against all specified derived objects' modification times.

28.6. Chaining of Rules

Part of shape's job is to be versatile in finding ways to produce a requested object. There may be situations when there is neiter an explicit dependency nor an implicit dependency directly producing the requested object. If this is the case, shape tries to construct a production (dependency) path by appropriately combining the rules it knows about.

Let's consider the following example, where the target *sample* is requestet, and shape has the derivation knowledge described in a Shapefile.

```
%.zzz : %.src
    cp %.src %.zzz

%.yyy %.xyz : %.zzz : +(FILTER)
    cp %.zzz %.yyy
    cat %.zzz | grep $(FILTER) > %.xyz

%.xxx : %.yyy %.aux
    cat %.yyy %.aux > %.xxx

sample : sample.xxx
    cp sample.xxx sample
```

Provided, there exist source objects *sample.src*, and *sample.aux*, shape will be able to find a way to produce the requested target *sample*. The sequence of derivations is as follows:

(1) sample.src → sample.zzz

(2) sample.zzz → sample.yyy, sample.xyz

(3) sample.yyy, sample.aux → sample.xxx

(4) sample.xxx → sample

APPENDIX A

ShapeTools-1.4 Standard Attributes

Name	Meaning	Values
alias	version alias names	list of alias names, like “vadm-4.2pre7” or “ShapeTools-1.4”
atime	time of last access	e.g. “Tue Jan 14 18:47:06 1992”
author	user who saved a version	user@do.ma.in (domain name does usually not include the hostname)
cachekey	unique key for cached versions	compound numeric built from creation date, process id, and a serial number e.g. “740148430.18469.6”
clead	comment line leader symbol	dependent on file type e.g. “# ” for Shapefiles
ctime	time of last status change	as *atime*
Description	descriptive text for module	multi line text
dsize	size of delta to previous version in bytes	numeric
generation	major revision number	numeric
Header	RCS-style version header	text
Intent	change intent	multi line text
host	name of current host	e.g. “avalanche”
Log	cumulative descriptive entries of all versions from the first up to this one	multi line text
lock/locker	user who locks a history	as *author*
ltime	time of last lock transaction	as *atime*
mode	access pprotection	e.g. “–rw–r––r––”
mtime	time of last modification	as *atime*
name	name part of an object identifier	e.g. “foo” for “foo.c”
note	short note describing the changes in this version	multi line text
owner	user who owns the repository in which this version is archived	as *author*
pred	bound version identifier of preceding version	e.g. “foo.c[3.22]” or “n/a”
revision	minor revision number	numeric
rtime	last time when history was locked	as *atime*

Name	**Meaning**	**Values**
self	bound version identifier for this version	e.g. “foo.c[3.23]”
selfpath	bound version identifier for this version including path	e.g. “/usr/proj/sample/foo.c[3.23]”
size	size of the version in bytes	numeric
state/status	version status	symbolic integers (busy, saved, proposed, published, accessed, and frozen)
stime	time when the version was saved	as *atime*
succ	bound version identifier of successive version	as *pred*
syspath	pathname part of an object identifier	e.g. “/usr/proj/sample” for “/usr/proj/sample/foo.c”
type	suffix part of an object identifier	e.g. “c” for “foo.c”
unixname	UNIX file name of this version	e.g. “foo.c”
unixpath	UNIX file name of this version including path	e.g. “/usr/proj/sample/foo.c”
version	compound version number consisting of generation and revision number	e.g. “3.22”
vtime	version time, modification time for busy versions od save time for saved/cached versions	as *atime*
xpoff	pseudo attribute that turns off subsequent attribute expansions	none
xpon	pseudo attribute that turns subsequent attribute expansion on	none