A toolkit for software configuration management

Axel Mahler, Andreas Lampen
Technische Universität Berlin

Abstract

For almost ten years, *Make* has been a most important tool for development and maintenance of software systems. Its general usefulness and the simple formalism of the *Makefile* made Make one of the most popular UNIX† tools. However, with the increased upcoming of software production environments, there is a growing awareness for the matter of *software configuration management* which unveiled a number of shortcomings of Make. Particularly the lack of support for version control and project organization imposed a hard limit on the suitability of Make for more complex development and maintenance applications.

Recently, several programs have been developed to tackle some of the problems not sufficiently solved by Make. **shape**, the system described in this paper, integrates a sophisticated version control system with a significantly improved Make functionality, while retaining full upward compatibility with Makefiles. **shape**'s procedure of identifying appropriate component versions that together form a meaningful system configuration, may be completely controlled by user-supplied *configuration selection rules*. Selection rules are placed in the *Shapefile*, **shape**'s counterpart to the Makefile.

The **shape** system consists of commands for version control and the shape program itself. It is implemented on top of the *Attribute File System* (AFS) interface. The AFS is an abstraction from an underlying data storage facility, such as the UNIX filesystem. The AFS allows to attach any number of attributes to document instances (e.g. one particular version) and to retrieve them by specifying a set of desired attributes rather than giving just a (path−) name. This approach gives an application transparent access to all instances of a document without the need to know anything about their representation. So, it is also possible to employ different data storage facilities, as for instance dedicated software engineering databases.

The project organization scheme of **shape** provides support for small (one man), medium, and large projects (multiple programmers/workstation network).

Keywords: tools, software configuration management, version control, software engineering environments

1. Background

For almost ten years, *Make* [6] has been a most important tool for development and maintenance of software systems. Its general usefulness and the simple formalism of the *Makefile* made Make one of UNIX' most popular tools. However, with the increased upcoming of software production environments, there is a growing awareness for the matter of *software configuration management* which unveiled a number of shortcomings of Make. Particularly the lack of support for version control and project organization imposed a hard limit on the suitability of Make in more complex development and maintenance applications.

The notion *Configuration management* has been introduced by US military and government institutions for a set of management techniques dealing with the complexity (and costs) of very large development and maintenance projects. Triggered off by the surge of interest in programming environments during the last few years, the term *software configuration management* (SCM) made its way from the management domain towards the software engineering and development domain. Configuration management has been tried‡ to define as "the process of identifying the *configuration items* in a *system*, controlling the release and change of these items throughout the system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and *correctness* of configuration items. It is a discipline applying technical and administrative direction and surveillance to (a) identify the functional and physical characteristics of a *configuration item*, (b) control changes to those characteristics, and (c) record and report change processing and *implementation* status" [9]. Major subtopics included in configuration management are *change control, configuration identification, configuration control, configuration status accounting*, and *configuration audit*.

In the UNIX domain, Make and source control systems as SCCS [13] or RCS [15] are in widespread use as configuration management tools. When designing a new configuration management toolkit for UNIX, one has to have a very close look at what is already there. In the categories given above, the areas addressed by these tools are *configuration identification* and *change control*, the mainly technical aspects of SCM (in contrast to the other, more *management* related disciplines). The configuration management toolkit described in this paper stays in the technical area and attempts to solve some of the problems not

† UNIX is a trademark of Bell Laboratories.
‡ well, if you consider an ANSI standard a *try*

sufficiently covered by existing tools, while at the same time laying the groundwork for further developments that implement support for *managing* software projects. The toolkit consists of a dedicated version control system, and **shape**, a significantly enhanced Make program. **shape** has full access to the version control system, and allows the user to specify configuration rules, to control the selection process for component versions during identification, build or rebuild of system configurations. For the time being, we choose to be upward compatible with Make and thus to retain its concept of openness and versatility. Besides, this helps to make a new – potentially complex – tool easy-to-learn and easy-to-use for the large developer community who is familiar with Make.

Since integration of version control was a major objective for our system, we had to face the need for a document identification scheme that goes beyond the usual way of specifying name and type of a document. As a consequence, we began to design an *attributed filesystem* (AFS) introducing a much more generalized scheme for document identification. The AFS comprises concepts for version control, support of variants or status models for example. Furthermore, it helps to abstract from the particular underlying data storage system in such a way that it makes no difference whether it is an ordinary filesystem or a dedicated database system.

In the following section we make a serious effort to use Make and RCS effectively for configuration management purposes and give an impression of some typical SCM related problems that are not (or at least very hard) adequately to control with Make and existing source control systems. Section 3 takes a closer look at how things are improved by **shape**. We explain the new concepts of **shape** and how it works internally. In section 4 we conclude the discussion and and outline the prospects for future work. Samples for a Makefile and a Shapefile that support the discussed activities can be found in appendix A and B respectively.

The paper assumes that you are familiar with the concepts of Make and the Makefile.

2. Doing it with Make

Although the full power of Make's basic concept takes effect only in the UNIX environment, the program has been ported to or reimplemented on a considerable number of systems. Without Make, UNIX wouldn't be what it is today. However, in the last years a number of attempts have been made to improve the program in a number of ways, complete reimplementations have been done, and quite a lot of criticism has (respectfully, though) been expressed, of what is considered weaknesses of Make [2, 3, 7, 8, 10, 14, 16]. Often, the discontent is related to some kind of specialized new task that Make is put to work on, which it wasn't originally designed to perform. So, this kind of criticism can also be interpreted as some compliment for a very flexible tool.

In [16] it is argued that one of Make's more serious drawbacks lies in the *lack of standards* and *differences between versions*. Makefiles are frequently written in an ad-hoc fashion rather than carefully designed, as they should be. An ambivalent issue about Make is that one tends to use it without having to think about it. This situation is supplemented by a lack of education. There are only few tutorials and guides that explain how to make the right use of Make. Even experienced Make-users happen to be unaware of some of Make's features and abilities.

Despite proven deficiencies, there are only few absolute no-no's with Make. An experienced and sufficiently stubborn practitioner will (almost) always be able to invent some workaround that serves her particular need. This results from a consequent concept of openness and extensibility. The combination of Make with tools as *sh, awk, sed, grep, cpp* ... makes Make really a devil of a fellow. Nobody said that all the power is easy to use, though.

In the subsequent discussion we present quite a hard try to (well, *kind of*) integrate Make with RCS, one of the finest available source control systems, intending to create a tool environment suitable for basic software configuration management tasks.

We're addressing three common application scenarios that relate to basic SCM tasks:

1) *cooperating developers:* a number of programmers independently working on different parts of the same targeted system.
2) *system integrator:* a functional role that gathers the work results of cooperating developers and creates *official* configurations (releases). The integrator has *read-only* access to all sources.
3) *maintenance:* backing up formerly saved system configurations for the sake of bug fixing or incorporating customer specific modifications while development goes on at head the system components' main lines of descent.

2.1. A Sample Project Setup

For the first scenario we assume that every programmer has a dedicated directory where the workfiles (we call them *busy versions*) of those components, she is in charge of, are kept. During component development these busy-versions undergo frequent changes. In order to test a work result, the programmer needs to build a complete system configuration from these *privately modified* components and *all remaining* components, possibly controlled (owned, reserved, locked) by other programmers. To avoid interference by temporarily inconsistent components, taken care of by other programmers, it is necessary to maintain consistent (or *milestone*) versions of these components and make them available. Making use of RCS, it's quite simple to organize mutual access to saved, (hopefully) consistent versions of all system components. RCS allows to store document revisions in archive files residing in a special subdirectory, RCS. To circumvent confusion with lots of different pathnames (of all the programmers' work directories) and to provide shareability of the Makefile, it is a good idea to create one single directory as repository for all the RCS archive-files, and let all members of a project have a *symbolic link* (named RCS) to that directory. Unfortunately, this works only with the BSD flavor of the operating system.

To arrange Make and RCS, a couple of new transformation rules, handling dependencies of sources from source archives must be added to the Makefile. A sample Makefile that does the job described here can be found in appendix A. These rules in conjunction with the setting of special macro *VPATH* which extends the scope wherein filenames (in this particular case their *suffixes*) are looked up, cause all .o files which's source is not present in the current directory to be generated from the corresponding, latest saved .c-source in the archive file. Temporarily retrieved sources are deleted immediately after the compilation is finished. The resulting object files, however, are kept and only regenerated when the corresponding source archive is touched, i.e. a new source version becomes available.

An important point in this approach is that cooperating developers have to trust each other in that new module versions are made available with *care*. The described scenario works quite well in preventing programmers from accessing components that are under immediate construction. This does not imply however that *functional changes* or *interface modifications* are properly advertised. In fact, herein lies a much more severe problem, because programming languages like C don't provide detection of these cases. Usage of *lint* would be advised here but for obvious reasons, routine use of lint is – to say the least – unpopular. Some mechanism for either accepting/rejecting of newly submitted versions or incremental testing with associated status control would be helpful.

Eventually needed include files must remain checked out, because Make doesn't allow to express that a derived object depends on (possibly multiple) sources stored in an archive. This might be considered harmless in the context of C but causes considerable overhead when applied to programming languages as Modula-2 or Ada, where each compiled object depends on at least two sources.

Another problem for cooperating developers is to keep their Makefiles consistent. Each programmer needs her own copy of the overall Makefile which should be *identical* for all programmers. Changes to Makefiles, e.g. introduction of new system components, new dependencies or different compiler options, should be published as soon as possible in order to maintain a common level of information. However, most of the changes made by a programmer will be very small, and therefore it is unnecessary to force her to edit – and maybe damage – the entire Makefile, a possibly huge, complex and extremely sensitive piece of information. For cooperative projects, it would be highly desirable to have such things as *distributed* or *modular* Makefiles providing localization of information while at the same time maintaining consistency with the whole of the project.

2.2. Making Releases

The creation of releases is the *system integrator's* job. Releases are system configurations, intended to be passed to the outside world or representing internal milestones. Releasing a product includes taking the (at least moral) responsibility for its functional integrity and to be prepared to react on maintenance requests. Establishing a release baseline means to *synchronize* the development stages of all components that are part of it and creating a *checkpoint* where all independent lines of development intersect and are bound together (i.e. make sure that all program modules work and cooperate properly, and all the documentation is up to date). We're talking testing and reading here, things that require a consciousness of responsibility, and mostly have to be carried out manually. The release process can be fairly complex: work results of programmers and writers might have to be reviewed or evaluated against specifications, unsatisfying results have to be rejected, corrected, evaluated again a.s.o. There is a complex relationship between software configuration management activities such as version control, version status accounting, and configuration auditing and systematic software quality assurance. An in-depth discussion of this matter can be found in [4].

To fix a release requires precise identification of the product configuration. For the sake of maintainability any given release should be exactly reproduceable. This comprises looking up all the source document versions and rebuilding any derived product under the same prerequisites as has been done originally. Although it is impossible to do this a 100% perfect, this can be accomplished to a certain degree by logging all characteristics of a configuration in a configuration identification document (*CID*). To meet minimal requirements, a CID should contain the following information:

- a complete list of the system components with version identification
- the version of the Makefile
- the current date, and the identification of the programmer in charge for the configuration
- identification of all tools and their versions that were involved in building the configuration.

Doing this with Make and RCS is not the easiest thing to do. When dealing with these requirements, it becomes obvious that the functionality of RCS and Make should be truly integrated rather than artistically woven. RCS provides for marking document revisions with state attributes (e.g. "released" and furthermore allows to associate unique symbolic names (e.g. "release2.1") with a revision. Such a symbolic name would typically be shared by all component revisions that are part of a given configuration. The problem is, that RCS is concerned with individual documents rather than (complete) sets of related documents and offers no way to make use of the information stored in the Makefile. Also, the question comes up, how to define the version number of a configuration, i.e. a set of programs, each composed from a number of modules, all with different revision numbers. It would be nice if RCS operations as `ci` could be applied to entire configurations, i.e. *targets* in the Makefile.

Make on the other side, having no idea what release building is all about, has its problems assuring that all components are at least saved and properly marked. Flexible handling of *variants*, be they implemented as conditional compiles, separate files hidden in subdirectories, or variant branches in RCS archives is also a nightmare with Make. When subsequently building different variants of a system, Make either doesn't detect that some files have to be recompiled due to change of compile-flags, or it recompiles all sources from scratch because it is unable to figure out if a particular module is actually *affected* by such a change.

In the sample Makefile from appendix A, when making the target "release", it is made sure, that all components are saved and systematically marked† before the programs are built and the CID is generated. To understand how the complete mechanism works in detail is left as an exercise to the reader.

2.3. Maintaining releases

The third scenario comprises the functions involved when modifications on releases shall be done. These are typically bug fixing, or customer specific tailoring of the released system.

The *maintenance engineer* begins her work with recovering all source versions belonging to the release in question. In order to avoid collisions with the current development versions, this has to be done in a directory especially created for that purpose. This directory should have an RCS subdirectory holding *copies* of the system master source archives. With the two added rules for handling RCS archives, Make is able to check out the appropriate source file versions for a given release name.

During reconstruction of the system it becomes evident, whether the *system integrator* has done a good job. If not all source versions belonging to the release are properly marked, the maintenance engineer is really in trouble. In this case, reconstructing of a formerly consistent configuration will be extremely time consuming. Obviously, a reliable release rebuilding mechanism is essential for the maintainers work.

The process of maintaining an old release causes forking of a new line of development on the basis of obsolete component versions. Depending on the complexity of the change request, a maintenance phase might last quite a while and involve a number of developers. The workspace for the maintenance process, however, should be separated from the main development area to avoid mutual interference. Following the described approach *without copying* the master source archives would cause unnecessary check out operations and recompilations in the main development area, each time a new revision is appended to the maintenance branch (RCS file is touched). Even worse, recompilations with the latest *main stream* revision would occur in the maintenance area, each time a new revision is checked in on the main line of development.

Besides the need for physical separation of the maintenance and development archives (resulting in organizational and diskspace overhead), the check out rule in the Makefile (`.cv.c:`) has to be rewritten in a way that the latest revisions from the *maintenance branch* are checked out instead of the latest revision on the main line of descent, which is RCS' default. Although it is *possible* to write such rules, this would further increase the complexity of the Makefile while decreasing the performance of the Make process. Creating completely new maintenance archives with the components of the maintained release as initial revisions, would result in inconsistent numbering between maintenance and development versions. A later reincorporation of maintenance branches into the master archives would be impossible.

Although this gives an idea of what can be done with Make‡, we believe that it also becomes obvious why the *do-it-with-Make* approach comes to its limits here. Makefiles of the given kind become extremely hard to write and maintain, are error-prone while the implemented function is still unsatisfying, and react very fragile to any kind of exception. While the first described scenario is still comparatively well supported by the presented approach, the introduction of more complex SCM activities puts a heavy load on Make and an even heavier burden on the Makefile writer.

For professional software production, SCM should be applied on a routine basis. Taking this into consideration, the described scenarios are everything but exotic. Things *have* to be much easier to use, more systematic, and much more robust.

3. How things can be improved

One of the most frequently denounced drawbacks of Make is its inability to react effectively on changes in the transformation environment, e.g. if a compile flag is changed. In discussing this matter, it has often been argued, that Make's restricted understanding of dependency as *time* dependency is the reason for this unsatisfying behavior. However, a second – closer – look at the problem leads to the conclusion that this is a mere *symptom* for the real problem, lying in the inability of the UNIX filesystem to handle more file attributes than those stored in the inode and the directory.

For the realization of **shape**, we tried to overcome the limitations imposed by the UNIX filesystem by creating the *attribute filesystem* interface (AFS) that provides an extended view of documents to application programs. This view comprises the concept of *document histories*††, as well as the possibility of tagging any number of *user-defined attributes* to document versions or complete document histories. Interpretation and use of these attributes is left to the application that uses them. Each *document instance* (i.e. one single version) is understood as a complex of *content data* and a set of *associated attributes*.

† marking accounts for unnecessarily compiling the complete system, because the archives are touched.

‡ I forgot to mention that some minor modifications had to be made in `rlog` (→ `"newrlog"`) and `ident`. `newrlog -y` returns nothing but the latest revision number of the given file.

†† represented as sequences of changes (deltas) [12].

Document attributes have the general form *name=value*, where *name* represents the attribute name, and *value* a possibly empty attribute value. There are a number of implicitly defined *standard attributes* for every document instance. Some of the standard attributes (e.g. name, size, owner, or protection attributes) are inherited from the UNIX filesystem, others (e.g. revision number or state) are AFS specific.

With AFS, documents are retrieved by specifying an *attribute pattern*. An AFS retrieve operation results in a − possibly empty − set of *document keys*, each of which representing a unique document instance that matches the specified attribute pattern. For the identification of documents, all attributes are considered equally suited. For instance, it is possible to retrieve all documents with an attribute *name=xyzzy*, or all documents with the attributes *author=andy* and *state=published*. The attributes that together form the *version id* guaranteeing the unique identification of one document instance, are under strict control of the AFS. These attributes are *document name*, *document type*, *generation number*, *revision number* and the *variant name*. Generation numbers are used to indicate major development steps that are common to all components of a configuration or subconfiguration. The revision number serves as an update sequence number for individual components within a generation. Generation number and revision number together are often referred to as *version number* (e.g. 3.0, 18.49).

The data presented and accessed by means of the AFS is stored in a data storage system such as the UNIX filesystem or some database. In the current implementation, the AFS abstracts from a set of *archive files* used to host contents and attributes of stored objects. Every regular file can be viewed as an AFS document, even if it has never been touched by an AFS application. In this case, it will be treated as a busy version without AFS specific or user-defined attributes. If such a file is checked into the version control system for instance, a source archive file will be created, and the missing standard attributes will be supplemented in a meaningful manner.

The fundamental difference between Make and **shape** is that Make is naive about versions of objects and **shape** is not. Make assumes that any object it deals with is a UNIX-file and therefore has only one version — the *current*. When evaluating dependencies, Make looks just at *name* and *modification time* of files. This is as good as Make can be on the basis of the UNIX filesystem. The fact that auxiliary tools like RCS had to be introduced in order to provide for multiple revisions of documents without creating name conflicts is another symptom, pointing at the same deficiency of the filesystem. RCS supplements a limited number of attributes such as revision number or state to the set of standard file attributes. Most of RCS' functionality is controlled by these attributes. However, there is no way to use these attributes to extend Make's idea of dependency, and integration of both tools has the character of a *'hack'*.

shape has an understanding of source versions and possibly multiple instances† of *derived objects* that exist at the same time (e.g. objects compiled from different versions of the same module). When producing a target, **shape** associates a number of *derivation attributes* with the resulting derived objects that *sufficiently describe* the current transformation. To describe a transformation *sufficiently* means to record all prerequisites that are needed to reproduce an identical copy of the derived object. Currently, these attributes are the *version ids of all source-objects* and the *flag definitions* that were in effect for the transformation.

While Make bases its decision whether to fire a transformation on the build-in *time-dependency-relation* between the target and its dependents, **shape** checks the target's derivation attributes against the attributes that a new transformation would generate. This causes **shape** to recompile a system if for instance compile flags have been changed, or other but the default (e.g. *newest*) source versions shall be used (for example in case any of them doesn't work as expected and couldn't be integrated). Also, recompilations can be avoided if an instance of a derived object can be found that already has the attributes that would be assigned to a newly produced one. This becomes interesting when more than just one instance of a derived object is kept and a different, previously saved (therefore immutable) source version shall be used. The **shape** toolkit offers support for multiple instances of derived objects stored in a cache-fashion administered *derived object pool*. The concept of derived object pools is borrowed from DSEE [1], a highly sophisticated software engineering environment running under Apollo's AEGIS operating system.

3.1. Elements of the toolkit

Besides facilities for keeping multiple instances of source− and derived objects, the **shape** toolkit provides programs for

- *basic version control*, such as `save` for creating inalterable versions, `retrv` to replace the current busy version by some formerly saved version, `vl`, `vinfo`, and `vcat` to browse information about document histories and view particular versions.
- *project interaction* that are used to organize and control the document submission process. `resrv` (developer) attempts to reserve the update privilege for a document history, `sbmit` (developer) submits a work result to be included into the next release of a system, `accpt` (system integrator) accepts a subitted document version and gives it an official, publically accessible state, and `rject` (system integrator) denies a submitted work result the official state and sends a corresponding message to the submitting programmer.
- *building configurations*, namely the program `shape` which builds a system configuration from a *Shapefile* or a *Makefile*.

† it should be noted that only source-objects are subject to version control and bear *version ids*. Derived objects do not have version attributes. Multiple instances of them are solely kept for the sake of efficiency, i.e. to prevent recompilations.

All programs of the toolkit are implemented on top of the AFS interface and share the same basic concept of document identification. The version control commands can be applied to entire configurations or subconfigurations, because they use the information stored in the Shape− or Makefile (actually, some of the version control programs are *links* to `shape`). In the remaining part of this article we will fully concentrate on the way configurations are built by **shape**.

3.2. The Shapefile

As it is the case with Make, everything that **shape** may do is controlled by a description file, the *Shapefile*. The **shape** program is upward compatible with Make and thus understands Makefiles, which are taken as system description, if no Shapefile can be found in the current execution context (which *can* be more than just the working directory if a project context is active). **shape** behaves identical to Make if a conventional Makefile is interpreted. Significant improvements with respect to Make are the introduction of *selection rules* and *variant definitions* into the description file to control the interaction with the attribute filesystem. Selection rules and variant definitions are expressed differently from the Makefile syntax. The formalism for the definition of default transformation rules has also been extended.

Selection rules control the selection process for component versions during identification, build or rebuild of system configurations. Variant definitions help to deal with the complexity of variant administration.

The Shapefile consists of four main components:

- system description
- selection rules
- transformation rules and
- variant definitions.

The syntax for the system description part is the same as for Makefiles. The remaining Shapefile sections have a slightly different syntax from Makefiles and must be preceded and ended by special comment sequences (`#% RULE-SECTION, #% VARIANT-SECTION, #% END-RULE-SECTION` a.s.o.).

3.3. Configuration Selection Rules

A selection rule is a named sequence of *alternatives*, separated by semicolons, which form a logical OR expression. Each alternative consists of a sequence of *predicates*, separated by commas, which form a logical AND expression. A selection rule succeeds if one of its alternatives succeeds (i.e. leads to the unique identification of some document instance).

A selection rule that − when activated − would cause the configuration of an experimental system (*"select the newest version of all components that I am working on; select the newest published version of all other components"*) might look like:

```
exprule:
    *.c, attr (author, $(LOGNAME)), attr (state, busy);
    *.c, attrge (state, published), attrmax (version).
```

Another example illustrates how known versions of particular modules could be configured into otherwise experimental systems:

```
special_rule:
    afs_def.h, attr (version, 8.22);
    afs_hparse.c, attr (version, 8.17);
    *.c, attr (author, $(LOGNAME)), attr (state, busy);
    *.c, attrge (state, published), attrmax (version).
```

In alternatives, the first predicate (e.g. `*.c`) is usually a pattern against which the name of a document that has to be retrieved is matched. The patterns used by **shape** have the same structure as those used by `ed(1)`. The other predicates allow to express certain requirements with respect to the attributes of documents. *Attr* and *attrge* are predefined predicates that require a specified attribute to be equal or greater-than-or-equal (with respect to a defined or *natural* order) a given value. The similarly predefined predicate *attrmax* requires documents to have a maximal value in the specified attribute. To provide a limited support for handling of *variants*, **shape** also has build-in predicate *attrvar* which affects the setting of some special macros, described later. In order to identify exactly one document instance by evaluation of a selection rule, the alternatives must be sufficient to single out one element from a possible set. Usually, the last predicate of an alternative should guarantee a unique selection. Predicates like *attrmax (revision), attr (state, busy),* or *attr (version, 4.2)* are examples of such selections.

shape checks whether the target already exists and is *current*, or tries to produce it from its *dependents*. A target is considered current if neither the source nor the production context have changed. A principal role for the retrieval of documents is played by their name attributes. In order to check whether a given target is current, both the target document itself, and all of its dependents, must be configured. A name† that has to be configured is passed as implicit parameter to the active selection rule.

† Conceptually, name *and type* of a document are passed as implicit parameters. In this discussion, *name* stands for the concatenation *name.type*

During the evaluation of the rule, the name is sequentially matched against the name pattern of the rule alternatives. If a pattern matches a name, **shape** attempts to complete the following predicate sequence. If an alternative fails, **shape** will go on and try the next alternative until the rule is completed.

After a name match, **shape** performs an AFS retrieve to initialize an internal *hit set*. The subsequent sequence of predicates will be applied to the set of found AFS objects. An alternative fails if one of the following conditions is true:

- the pattern does not match
- a predicate of the alternative fails
- the cardinality of the hit set is not equal to one, i.e. no unique name resolution was possible.

The application of predicates to the hit set results in the removal of all AFS objects that do not conform to the specified requirements from the hit set. A predicate's evaluation fails if the cardinality of the hit set becomes equal to zero. After the target and all of it's dependents have been configured, the source version ids and the transformation environment are evaluated against the derivation attributes of the derived objects (if present) that would have to be produced.

Selection rules can be activated on a per production basis by simply giving the name of the rule as the first dependent of a production. Thus, it is possible to define targets for the configuration of e.g. test systems and releasable systems within the same **shape**-file.

```
test: exp_rule prog
release: rel_rule prog
prog: x.o y.o z.o
```

A selection rule remains active until it is superseded by activation of another selection rule or until the target that caused the activation is produced. If no selection rule is specified, the default rule, which is the same as Make's ("*select the busy version in the current directory*"), is active.

The following pictures give an overview of the **shape** process.

boxwid = 0.6i; boxht = 0.6i; circlerad = 0.3i; Pa: box "parse" "**shape**" "file" arrow <- up 0.3i from top of Pa circle "**shape**" "file" arrow right 0.2i from right of Pa circle "name/" "type" arrow right 0.2i Ru: box "rule" "interpreter" arrow <- up 0.3i from top of Ru circle "selection" "rules" arrow right 0.2i from right of Ru circle "set of" "required" "attributes"; arrow right 0.2i box "AFS" "retrieve"; arrow right 0.2i circle "filekey"; arrow right 0.2i box "AFS" "get" "attributes"; arrow right 0.2i circle "complete" "set of" "attributes"

Fig. 3.1: Identification

After having uniquely identified the system's components, the necessary transformation actions can be performed.

boxwid = 0.6i; boxht = 0.6i; circlerad = 0.3i; Sa: circle "source" "attributes" move down 0.3i from Sa.b; Ta: circle "target" "attributes" move down 0.15i from Sa.b; move right 0.6i; Va: box "target" "valid?" arrow from Sa.r to Va.nw arrow from Ta.r to Va.sw arrow right 0.3i from Va.r circle "yes/no" arrow right 0.3i Pr: box "produce" arrow <- up 0.3i from Pr.t circle "production" "rules" arrow right 0.3i from Pr.r circle "new" "target" arrow right 0.3i box "AFS" "set" "attributes" arrow right 0.3i circle "attributed" "target"

Fig. 3.2: Production

The search space for retrieve operations is either defined by the *current project* (an explicitly selected work context), or by an environment variable that describes the search hierarchy in a fashion that depends on the underlying data storage system. If **shape** cannot find a document in a programmers workspace, it tries to connect to a *project server*, whose address is also defined by the current project. The project server might reside locally or somewhere in the network. It provides controlled access to the project's database. Thus, if a particular document could not be tracked down locally, it might still be somewhere in the project library.

3.4. Transformation Rules

It is part of Make's philosophy, that a transformation produces exactly *one target at a time*. As long as all instances of source and target objects are versionless, there is no big problem with *side effects* (creation of *additional* objects not specified by the transformation rule — TEX for example, creates a transscript and an auxiliary file along with the dvi-file), because resulting files are automatically updated. Make, just looking at name and modification time wouldn't reproduce these files, if subsequently any dependency on them should be encountered. **shape**, however, basing its decision whether to activate a transformation on a bigger set of attributes, has to *explicitly mark* each produced object with the attributes describing the characteristics of the transformation. In order to avoid overhead transformations, **shape** needs to know the complete set of resulting objects.

To give **shape** a chance to find out about a transformation's significant characteristics, it was necessary to change concept and syntax of transformation rule definitions with respect to Make. In particular, **shape** needs information about the flag− (macro−) definitions that affect the transformation, and names of all objects that are going to be produced. Make's syntax for specifying a default transformation rule doesn't provide for either information. For **shape**, we choose to employ a definition syntax, similar to that used by *cake* [14]. Cake's transformation specifications are based on *name variables*. For example, a possible rule for compiling C programs is

```
%.o: %c
     cc −c %.c
```

with '%' as variable symbol. Transformation specifications are actually *templates* for an infinite number of dependencies, each of which is obtained by consistently substituting a string for the variable '%'. When an object has to be produced, **shape** determines which transformation rule applies, by matching the object's name against all tokens to the left of the colon. In this matching process, '%' is treated as wildcard character. Once a matching token has been found, the '%' character is consistently substituted throughout the transformation rule. In the example above, a request to produce an object *matchit.o* would result in a virtual Shapefile entry

```
matchit.o: matchit.c
        cc −c matchit.c
```

The resulting set of source and object names is taken to be the *specification* of the transformation carried out by the shell script associated with the rule. **shape** doesn't know, what is actually done by the shell script, but *assumes* that *all specified objects will be produced* by the transformation, each depending on all source objects and relevant flag− and macro definitions. Another element of transformation rule definitions that hasn't been mentioned yet, is the specification of the macros that are applied in the rule's shell script. To specify them, the rule specification is extended by another, optional colon which separates the flags and macro names from the source names. Thus, a complete transformation rule definition to create a .o object from a yacc source looks like

```
%.o : %.y    : $(YFLAGS) $(CFLAGS)
     $(YACC) $(YFLAGS) %.y
     $(CC) $(CFLAGS) −c y.tab.c
     rm y.tab.c
     mv y.tab.o %.o
```

3.5. Configuration Identification and Reproduceability

shape can, for the purpose of *configuration identification*, be asked to produce a configuration identification document (CID). CIDs as produced by **shape** include:

- all components' version identification (composition list)
- the components' variant identification (if present)
- the version of the Shapefile
- the *project domain*, in particular the projectname and the next-higher domain name in which the projectname can be resolved (e.g. a network-, host- or pathname)
- identification of all tools and their versions that were involved in building the configuration (this applies − of course − also to **shape**)
- all macro definitions that were imported from the environment
- the current date, and
- the identification of the programmer who created the configuration.

In order to provide for large configurations that consist of more or less independently maintained subconfigurations, CIDs may also contain references to corresponding *sub*-CIDs. CIDs are the ultimate and most precise description of a configuration. They are designed to describe the circumstances of the configuration identification and build process completely. CIDs are themselves derived objects containing more information than the derivation attributes described above are able to hold. Because this information is crucial for the maintenace of releases, CIDs are subject to version control. The version id of the CID defines the version id of the described product configuration.

Complete reproduceability of system releases is a goal that is hard to achieve. To rebuild old configurations as identically as possible does not only require to keep all document versions that have ever been part of a release, but also a straight history of all tools (e.g. compilers) that have ever been used to produce releases. The cost for this might be considerable administrative overhead, because every project needs a current *resource registry*, where all production tools are logged.

shape has a built-in *rebuild function*. Rebuilding of system configurations is a prerequisite for product maintenance, in particular bug-fixing and customization. For the rebuilding of a release, **shape** may be fed with a CID. It will retrieve the identified **shape** file version and start the system build based on the component versions specified in the CID and the system architecture described in the **shape** file. Whether **shape** will complain about mismatching tool versions, will depend on the grade of accuracy to which a rebuild is done. We believe that this − like a couple of other aspects − should be subject to customizations.

3.6. Controlling variants

There is a common understanding of variants as *alternative* realizations of the same concept. Each alternative has its own revision history, and so there is not necessarily a temporal relationship between them. The typical example for the genesis of variants is porting of software products to different architectures. The idea of variants sounds quite simple, but to actually handle them can be an extremely difficult task.

In C, the concept of variants is most frequently implemented by marking code sections to be conditionally compiled, depending on definitions supplied in a special header file or as command line flags from within the Makefile. For other programming languages, such as Modula-2, variant modules have to be physically separated† and must be stored in different directories. Both techniques are understood by **shape**. To provide means for systematic variant control, **shape** allows to define variant names. These are associated with variant-flags that will be passed to the transforming tool, and a variant-path that extends the search space by directories which could host a document variant. The following example illustrates the use of variant definitions:

```
#% VARIANT-SECTION
vclass system ::= (vaxbsd, munix)
vaxbsd:
     vflags="-DUNIX -DBSD42 -DSTDCC -DPSDEBUG"
     vpath="sys/vaxbsd"
munix:
     vflags="-DUNIX -DSYS5 -DPCSCC -DNOVAX -DPSDEBUG"
     vpath="sys/sys5"
debug:
     vflags="-g -DDEBUG"
#% END-VARIANT-SECTION
```

Variant names, as defined in the variant section, can be applied in selection rules by using the predefined selection predicate *attrvar*. They provide a unified concept for the administration of variants within the version control system, with preprocessor technique, and with physical separation. Configuration selection rules making use of **shape**'s variant handling might look like:

```
fsexp:
     afdelta.*, attr (state, saved), attrvar (munix),
          attrmax (revision);
     af*.*, attr (state, busy), attrvar (debug),
          attrvar (munix), attrvar (unixfs);
     make*.*, attr (generation, 4);
     *.*, attrvar (munix), attr (state, busy).
```

shape uses the special macros *vflags* and *vpath* to hold preprocessor options, and extensions to the default document search path. Locations specified by *vpath* are searched for documents prior to the default location. If a document with the specified attributes is found in a *vpath*-directory, the default directory will *not* be searched.

Initially, both macros have an empty value. When variants are selected in an alternative of a selection rule, the associated macro values are *added* to *vflags* and *vpath*. In our example, during evaluation of the second alternative of rule *fsexp* (`af*.*,`), *vflags* would become:

```
vflags="-g -DDEBUG -DUNIX -DSYS5 -DPCSCC -DNOVAX -DPSDEBUG -DFS"
```

and *vpath* would be:

```
vpath="sys/sys5:data/unixfs".
```

Upon completion of each alternative both macros are reset.

The construct *vclass system ::= (vaxbsd, munix)* defines a *variant class*. Variant classes define mutually exclusive variant names. Variant classes can usually be given meaningful names, because they mostly correspond to certain properties, in which the particular variants vary. A variant class *cputype* for instance, with element names *IBM4381, VAX630, VAX7XX, m68k* a.s.o. might be defined in order to prevent the selection of different cpu-specific modules with mismatching cpu attributes. **shape** will complain if such a condition is detected. **shape** encourages systematic description of variant properties in the variant section of the Shapefile. Flag– or macro definitions that are related to variants should be defined in this section rather than places like *CFLAGS*. So far, this is the only concept in **shape** for ensuring semantic consistency across variant system components. In this area, more work needs to be done.

---

† you can, of course, also use `cpp` or `m4` . . .

4. Conclusion

At first glance there seems to be no big difference between the functionality of **shape** and a reasonable integration of Make and RCS. However, by integrating version control and system building, we were able to eliminate a number of problems that are extremely difficult to handle with Make and RCS. Each element of the **shape** toolkit utilizes the available data (e.g. "save" operations can be performed on whole systems; production steps can be described more precisely). Furthermore, the implementation of the AFS interface provides a well defined entry point to the version control system that can also be used by other than the toolkit's application programs. In fact, we believe that the AFS facility is general enough to be used in completely different contexts but version control.

The most visible difference between **shape** and Make/RCS is the enriched Makefile – now called Shapefile – with special syntax for the definition of system variants and version selection rules. With the simplicity but expressiveness of its version selection rules, **shape** makes it easy to build and maintain consistent system configurations from a possibly huge number of different source document versions.

Nevertheless, there are still many edges where further work has to be done.

shape does not support dynamic dependencies; a feature that Make users often miss. Because inter-module dependencies can change between versions, the problem of version control and system building had to be tackled first. When adding support for dynamic dependencies, it is important to do this in a *programming language independent* way in order to maintain the openness and generality of the Make concept. We are thinking about a well defined interface to application supplied, language dependent *closure operators*. Some interesting results in this area have been presented with the *Adele* system [5].

Another important feature that **shape** still lacks, is a concept for modularity of Shapefiles. As this implies language concepts that go way beyond what current Shapefiles are able to express, we will discuss the issue in the context of a complete review of the concepts we choose for the first implementation of **shape**.

The variant support which is described more detailed in [11] has also only rudimentary character as it primarily aims at practical life support for the software developer. However, we feel that in this area a lot more work is due.

5. Acknowledgement

This work is part of the **UniBase** project. The project is supported by the Bundesministerium für Forschung und Technologie (Federal Ministry for Research and Technology) under grant ITS 8308.

References

1. APOLLO, *Domain Software Engineering Environment (DSEE) Reference,* Apollo Computer Inc., Chelmsford MA., July 1985.

2. AUGMAKE, ‘‘An Augmented Version of Make,’’ *System V/68 Support Tools Guide*, pp. 21-40, Motorola Microsystems Inc., Ord.No. M68KUNSTG/D1, January 1983.

3. Erik Baalbergen (erikb@cs.vu.nl), ‘‘Thoughts about ‘Make’ – Summary,’’ *comp.unix.wizards*, USENET, November 1987.

4. Edward H. Bersoff, Vilas D. Henderson, and Stanley G. Siegel, *Software Configuration Management,* Prentice Hall, Englewood Cliffs, N.J., 1980.

5. J. Estublier and N. Belkhatir, ‘‘Experience with a Database of Programs,’’ *SIGPLAN Notices*, vol. 22, 1, pp. 84-91, ACM, Palo Alto, California, December 1986.

6. Stuart I. Feldman, ‘‘MAKE - A Program for Maintaining Computer Programs,’’ *Software - Practice and Experience*, vol. 9,3, pp. 255-265, March 1979.

7. Glenn S. Fowler, ‘‘A Fourth Generation Make,’’ *Proceedings of the USENIX Summer Conference*, pp. 159-174, USENIX asc., Portland, Or., June 1985.

8. Andrew Hume, ‘‘Mk: A Successor to Make,’’ *Proceedings of the USENIX Summer Conference*, USENIX asc., Phoenix, Ariz., June 1987.

9. IEEE, *IEEE Standard Glossary for Software Engineering Terminology,* IEEE, New York, N.Y., February 1983.

10. David B. Leblang and Gordon D. McLean, Jr., ‘‘Configuration Management for Large-Scale Software Development Efforts,’’ *Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 122-127, GTE Laboratories, Harwichport, Massachusets, June 1985.

11. Axel Mahler and Andreas Lampen, ‘‘shape – A Software Configuration Management Tool,’’ *Proceedings of the International Workshop on Software Version and Configuration Control*, German Chapter of the ACM, Grassau, West-Germany, January 1988.

12. Wolfgang Obst, ‘‘Delta Technique and String-to-String Correction,’’ *Lecture Notes in Computer Science*, Springer Verlag, Berlin, September 1987.

13. Marc J. Rochkind, ‘‘The Source Code Control System,’’ *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 364-370, 1975.

14. Zoltan Somogyi, ‘‘Cake: A Fifth Generation Version of Make,’’ *Australian Unix system User Group Newsletter*, vol. 7, no. 6, pp. 22-31, April 1987.

15. Walter F. Tichy, ‘‘RCS - A System for Version Control,’’ *Software - Practice and Experience*, vol. 15,7, pp. 637-654, July 1985.

16. D.M. Tilbrook and P.R.H. Place, ‘‘Tools for the Maintenance and Installation of a Large Software Distribution,’’ *Proceedings of the USENIX Technical Conference and Exhibition*, pp. 223-237, USENIX Association, Atlanta, Ga., June 1986.

Appendix A: A sample Makefile

```
#  Transformation rule definitions and
#  configuration management related rules and macros


     @-if [ -s $(SRCDIR)/$*.c ] ; \
     then : ; \
     else \
     echo temporarily checking out $*.c --- $(VID); \
     (cd $(SRCDIR); $(CO) $(COFLAGS) $*.c) > /dev/null; \
     $(CC) -c $(CFLAGS) $*.c; \
     rm $*.c; \
     fi;

     @-if [ -s $(SRCDIR)/$@ ] ; \
     then : ; \
     else \
     echo checking out $@; \
     (cd $(SRCDIR); $(CO) $(COFLAGS) $@) > /dev/null; \
     fi;

SRCDIR = /u/shape/apps
VPATH = RCS
USERID  = `/usr/ucb/whoami`\@`hostname`   # should e.g deliver 'axel@coma'
TMPNAME = .scm-stuff

#  Tool definition & location part
#  (these macros might be site and/or system-dependent)

SHELL = /bin/sh
RCSPATH = /usr/local

CC = cc -DCFFLGS='"$$Flags: $(CFLAGS) $$"'
CI = $(RCSPATH)/ci
CO = $(RCSPATH)/co
RLOG = $(RCSPATH)/newrlog  # added option '-y' - print only revision number
IDENT = $(RCSPATH)/ident
RCS = $(RCSPATH)/rcs

NGFLAGS = -f -l -q -m"New System Generation" -s"Stable"
NRFLAGS  = -q -l -m"This version is part of a release" -s"Release"
NRCIDFLAGS = -l -s"Frozen"

COFLAGS = -q $(VID)
LOGFLAGS = -h

#  Configuration definition part

CFLAGS = -g -DUNIX -DVAX -DUNIXFS -DJOBCONTROL -I$(AFSINC)

#  Installation specific part

INSTALDIR = /u/shape/bin

#  Product definition part

COMPONENTS = save.c dosave.c retrv.c doretrv.c mkattr.c project.c sighand.c \
         util.c save.h retrv.h project.h afsapp.h save.1 retrv.1

PROG = shapetools
PRODUCTS = save retrv
```

```
VERSION = version

MYINC = save.h project.h
SAVEOBJS = save.o dosave.o
RETROBJS = retrv.o doretrv.o mkattr.o
COMMON = project.o sighand.o util.o version.o
ALLOBJS = $(SAVEOBJS) $(RETROBJS) $(COMMON)

AFSLIB = /u/shape/lib/libafs.a

AFSINC = /u/shape/src/inc

#  Component dependencies

all: save retrv

save: $(SAVEOBJS) $(COMMON) $(AFSLIB)
	cc -o $@ $(SAVEOBJS) $(COMMON) $(AFSLIB)

$(SAVEOBJS): save.h project.h afsapp.h

dosave.o: $(AFSINC)/afs.h

retrv: $(RETROBJS) $(COMMON) $(AFSLIB)
	cc -o $@ $(RETROBJS) $(COMMON) $(AFSLIB)

$(RETROBJS): $(AFSINC)/afs.h retrv.h project.h afsapp.h

$(COMMON): $(AFSINC)/afs.h afsapp.h

project.o: project.h

$(ALLOBJS): Makefile

install: all
	install -c -o axel -g unib save $(INSTALDIR)
	install -c -o axel -g unib retrv $(INSTALDIR)
	(cd $(INSTALDIR); rm -f Save; ln save Save)

#  These rules are all configuration management related, and serve to
#  prepare system generations and releases. It should not be necessary
#  to modify them for different products, as long as certain conventions
#  are obeyed: Define the main-product name in the PROG macro. List
#  all components' names (i.e. each revisable entity) in COMPONENTS.
#  Names of individually produced subtargets should be listed in PRODUCTS.
#  The main target should be 'all'.
#
#  This Makefile sample is intended for 'single Makefile systems'.
#  To apply the proposed SCM scheme to more complex systems, some
#  more rules and conventions have to be defined.
#

release: preprel logconf1 all logconf2

logrelease: logconf1 logconf2

generation:  # assumes that all COMPONENTS have a busy version
	@incr $(SRCDIR)/.genno
	@/bin/echo Declaring generation `cat $(SRCDIR)/.genno` for \
	$(PROG) system
	@rm -f .relno
	@-(cd $(SRCDIR) ; \
```

```
	$(CI) $(NGFLAGS) -r`cat .genno`.0 $(COMPONENTS))

preprel:
	@incr .relno
	@/bin/echo Preparing release `cat .relno` in generation \
	`cat $(SRCDIR)/.genno` of $(PROG)
	@-(objdir=`pwd` ; cd $(SRCDIR); $(CI) $(NRFLAGS) Makefile \
	$(COMPONENTS) > /dev/null 2> .diag ; \
	awk '{ if ($$2 == "warning:") print $$6; \
	if ($$2 == "error:") print $$5 }' .diag > .ddiag; \
	for i in `cat .ddiag`; \
	do \
	v=`$(RLOG) -y $$i`; \
	echo marking $$i\[$$v\] ; \
	$(RCS) -n"$(PROG)_v`cat .genno`r`cat $$objdir/.relno`":$$v \
	$$i >  /dev/null 2> /dev/null ; \
	done ; \
	for i in $(COMPONENTS); \
	do \
	if grep $$i .ddiag > /dev/null ; \
	then : ; \
	else  \
	v=`$(RLOG) -y $$i`; \
	echo marking $$i\[$$v\] ; \
	$(RCS) -n"$(PROG)_v`cat .genno`r`cat $$objdir/.relno`":$$v \
	$$i >  /dev/null 2> /dev/null ; \
	fi; \
	done ; \
	rm -f .ddiag .diag )

logconf1:
	@/bin/echo Conf-ID: $(PROG) version `cat $(SRCDIR)/.genno` \
	release `cat .relno` of `/bin/date`, > $(PROG).cid
	@/bin/echo 'char *version () { static char ConfID[] = \
	"'`cat $(SRCDIR)/.genno`.`cat .relno` '('`date` by $(USERID)')"; \
	return ConfID; }' > $(VERSION).c
	@$(CI) $(NRFLAGS) -n"$(PROG)_v`cat $(SRCDIR)/.genno`r`cat .relno`" \
	$(VERSION).c > /dev/null 2> /dev/null;
	@/bin/echo by $(USERID) >> $(PROG).cid;
	@(cd $(SRCDIR) ; $(RLOG) $(LOGFLAGS) Makefile $(COMPONENTS)) > \
	$(TMPNAME);

logconf2:
	@/bin/echo Logging configuration in $(PROG).cid;
	@(cd $(SRCDIR); $(IDENT) $(PRODUCTS) | awk '{ if ($$1 == "$$Header:") \
	printf ("     Header: %s %s %s %s %s %s\n",$$2,$$3,$$4,$$5,$$6,$$7); \
	else print $0 }') >> $(PROG).cid;
	@-$(IDENT) Makefile | awk '{ if ($$1 == "$$Header:") \
	printf ("     Header: %s %s %s %s %s %s\n",$$2,$$3,$$4,$$5,$$6,$$7); \
	else if (($$1 != "$$Flags:") && ($$1 != "$$Log:")) \
	print $$0 }' >> $(PROG).cid;
	@cat $(TMPNAME) >> $(PROG).cid;
	@echo
	@echo '***' DESCRIBE PURPOSE OR DESTINATION OF THIS RELEASE \
	'['`cat $(SRCDIR)/.genno`.`cat .relno`']' '***'
	@echo '(terminate with ^D or single '"'.'"')'
	@$(CI) $(NRCIDFLAGS) -r`cat $(SRCDIR)/.genno`.`cat .relno` \
	$(PROG).cid > /dev/null 2> /dev/null;
	@rm $(TMPNAME)
```

Appendix B: A sample Shapefile

```
#
#  Tool definition & location part

SRCDIR = /u/shape/apps
INSTALDIR = /u/shape/bin

SHELL = /bin/sh
CC = cc

#% RULE-SECTION

fsexp:
    af*.c, attrge (state, published), attrmin (state),
        attrvar (unixfs), attrvar (debug);
    *.c, attr (state, busy), attrvar (unixfs), attrvar (debug);

fsrelease:
    *.c, attr (state, frozen), attrvar (unixfs);

dbexp:
    af*.c, attrge (state, published), attrmin (state),
        attrvar (damokles), attrvar (debug);
    *.c, attr (state, busy), attrvar (damokles), attrvar (debug);

#% VARIANT-SECTION

vclass database ::= (damokles, unixfs)

damokles:
    vflags="-O -DDAMO -DUNIX -DVAX -DJOBCONTROL"
    vpath="data/damokles"
unixfs:
    vflags="-O -DUNIXFS -DUNIX -DVAX -DJOBCONTROL"
    vpath="data/unixfs"
debug:
    vflags="-g -DDEBUG"

#% END-VARIANT-SECTION

CFLAGS = -I$(AFSINC)

#  Product definition part

COMPONENTS = save.c dosave.c retrv.c doretrv.c mkattr.c project.c sighand.c \
         util.c save.h retrv.h project.h afsapp.h save.1 retrv.1

PROG = shapetools
PRODUCTS = save retrv
VERSION = version

MYINC = save.h project.h
SAVEOBJS = save.o dosave.o
RETROBJS = retrv.o doretrv.o mkattr.o
COMMON = project.o sighand.o util.o version.o
ALLOBJS = $(SAVEOBJS) $(RETROBJS) $(COMMON)

AFSLIB = /u/shape/lib/libafs.a

AFSINC = /u/shape/src/inc
```

```
#  Product dependencies

all: fsexp save retrv

release: fsrelease save retrv

dball: dbexp save retrv

save: $(SAVEOBJS) $(COMMON) $(AFSLIB)
	cc -o $@ $(SAVEOBJS) $(COMMON) $(AFSLIB)

$(SAVEOBJS): save.h project.h afsapp.h

dosave.o: $(AFSINC)/afs.h

retrv: $(RETROBJS) $(COMMON) $(AFSLIB)
	cc -o $@ $(RETROBJS) $(COMMON) $(AFSLIB)

$(RETROBJS): $(AFSINC)/afs.h retrv.h project.h afsapp.h

$(COMMON): $(AFSINC)/afs.h afsapp.h

project.o: project.h

install: release
	install -c -o axel -g unib save $(INSTALDIR)
	install -c -o axel -g unib retrv $(INSTALDIR)
	(cd $(INSTALDIR); rm -f Save; ln save Save)
```