An Integrated Toolset for Engineering Software Configurations‡

Axel Mahler and *Andreas Lampen*
Technische Universität Berlin, West-Germany

Abstract

Configuration management in toolkit oriented software development environments (SDE), such as the UNIX† system, is a long standing nuisance. Mostly, one has to face the choice between poorly or not at all integrated, independent tools, or highly integrated, most specialized, and often language dependent environments. The first choice offers very limited support for a complex task that needs a broad informational basis. The second choice often takes away the programmers' most cherished tools, forces him to adopt some different work discipline, and thereby eventually restricts his creativity. The toolset described in this paper integrates a dedicated version control system and shape, a significantly enhanced Make [Feld79a] program, on the basis of a common object model. This object model comprises multiple versions of software objects as well as conventional file system objects. Taking this approach made it possible to have a sufficiently integrated toolsystem for engineering software configurations while retaining the flexibility of the basic toolbox philosophy, permitting the use of 'off-the-shelf' tools, e.g. editors or compilers.

1. Introduction

Configuration management is a management technique, designed to control very large development and maintenance projects. As military and government institutions played a principal role in the establishment of configuration management, the related terminology is precisely defined in an ANSI standard [IEEE83a]. An in-depth discussion of the underlying concepts can be found in [Bers80a].

Triggered off by the surge of interest in programming environments (PE) 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. Today, there are a number of PEs with built-in SCM support (e.g. DSEE, Adele, ISTAR [Lebl84a, Estu86a, Dows86a]). However, it is not clear whether SCM is more of a development or management task. In particular, we suspect that strongly management-oriented approaches to SCM are likely to produce red tape, tending to stand in the (programmer's) way, and consequently discourage their systematic application. On the other hand, strongly development-oriented SCM tools often perform a brilliant job in supporting the programmer, but produce little or no usable information for project management. To avoid confusion of management-oriented and development-oriented interpretation of SCM, we'd rather prefer the term *software configuration engineering* (SCE) to emphasize the more technical aspects as version control, configuration identification, and system building.

Our toolsystem for engineering of software configurations consists of an *object base* for attributed software objects, a dedicated *version control system*, and shape, a significantly enhanced Make program. Enhancements include full access to the object base, and support of *configuration rules* which control the selection process for component objects during identification, build or rebuild of system configurations.

Since integration of version control was a major objective for our system, we had to face the need for an object identification scheme that goes beyond the usual way of specifying just name and type. As a consequence, we began to design an object base for attributed software objects [Lamp88a] that allows to attach any number of attributes to software objects, and introduces a much more generalized scheme for object identification. The object base's name, *attributed filesystem* (AFS), mirrors the basic idea, to enhance the regular UNIX filesystem by supporting more fileattributes than just a name and the information stored in the inode. The AFS comprises concepts

‡ in *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments* pp. 191-200, SIGSOFT SE Notes Vol. 13-No. 5, SIGPLAN Notices Vol. 24-No. 2, Boston MA, 28-30 November 1988
† UNIX is a trademark of Bell Laboratories.

for version control, support of variants or status models for example. Furthermore, it helps to abstract from the particular underlying data storage system. For the AFS application writer it makes no difference whether it is an ordinary filesystem or a dedicated database system like e.g. DAMOKLES [Ditt86a].

1.1. Related Systems

At first glance, the user-level appearance of the shape toolkit resembles a combination of Make and a version control system such as SCCS [Roch75a] or RCS [Tich85a]. Over the last years some unsatisfying attempts have been made to integrate Make with source code control systems [Nick83a, AUGM83a]. The CMS/MMS tool [DECC84a, DECM84a] under VAX/VMS corresponds to *Augmented Make* and SCCS. CMS and SCCS not only provide a user command interface, but also give application programs direct access to the version control facilities through a procedural interface. However, despite availability of these interfaces, neither Make nor MMS are build on top of them and more complex configuration management applications find a hard limit in this lack of integration. shape's AFS interface can be seen as an extended and more generalized interface to the version control system. shape and the related version control commands are entirely build on top of this interface.

Besides these toolbox oriented systems, there are some highly specialized configuration management tools that are usually part of a software engineering environment. Because they are part of an overall tool concept that also includes programming languages, the functionality of these tools is often more comprehensive in terms of consistency enforcement. Cedar's System Modeller [Lamp83a] and the Gandalf version control facility [Habe82a] fall into this category. An interesting approach to the determination of system configurations based on sets of software objects and functions to model the relations between them is presented in Jasmine [Marz86a].

Common to all of the systems mentioned above is that a system description document (*system model*) is employed to define components and overall architecture of a software system. In contrast to that, the Adele configuration manager *derives* a configuration from a system description in terms of *desired characteristics* rather than some sort of component list. The description is evaluated against a *module dependency graph* and *manuals* (module descriptions) stored in a database of programs. The Odin System [Clem86a] in turn uses a (compiled) knowledge base, called *derivation graph*, describing derivative relationships between *document types*, and associated tools to perform necessary derivations in order to obtain a requested object. Odin integrates remarkably well with the standard UNIX tool environment and can be used to construct comprehensive special purpose environments from existing tools [Wait88a].

Finally, DSEE represents to the authors' knowledge the most outstanding configuration management system so far. It is based on system models, describing the version-independent structure of a software system and *configuration threads*, describing the version requirements that must be met by the elements of a configuration to be identified. DSEE combines configuration management with sophisticated project support facilities, enabling large programmer teams to cooperate over a (local or wide area) computer network.

In the design of shape, we tried to adopt a number of the best concepts mentioned above. Namely Make, Adele and DSEE had a strong influence on the design of the shape toolkit. Despite the integration of the toolkit itself, care has been taken to smoothly fit the SCE system into the standard UNIX environment. The entry cost for new users (in terms of initial learning effort) is very low. The system may be used as just a friendly version control tool, an upward compatible replacement for Make, or an intelligent integration of both. Utilization of the toolkit can be gradually extended up to a network based project support environment, employing a number of programmer teams working on large software development and maintenance projects.

The simple, nature of the toolkit's basic user interface offers a broad range of possibilities to create dedicated user interfaces that provide an integrated, problem oriented view of the SCE system. Currently in the works are an implementation of a specially tailored SCE shell on the basis of [H87a] (a graphical user interface system), and an implementation of a SCE-tool for the *X Window System*.

The rest of this paper describes the shape program and AFS, the object base on top of which shape and the version control commands are built. Also, a number of problems, encountered while trying to introduce general support for the handling of *variants*, and implementing AFS on top of DAMOKLES, will be discussed.

2. The Shape Toolkit

The user interface of the SCE toolkit consists of a set of commands for version control, project interaction, and the shape program itself. The appearance of the version control commands is not unlike to RCS' user interface. There are funtions to save and restore an object version (`save/retrv`), to manipulate object histories and attributes of object versions (`vadm [ -delete | -setattr | -chmod` (etc.) `]`), as well as a funtion to obtain information about the contents of the object base (`vl`, gives a `ls(1)`-like view of the object base).

The toolkit supports a basic project organization scheme that distinguishes private workspaces for individual developers, and a central library that is shared by the project team. The project library is administered by the system integrator. The toolkit provides functions for project interaction, controlling concurrent development (*object locking*), and orderly submission of software objects. `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 submitted 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.

Version control and project interaction are functionally supported by a built-in general version status model. The states `busy`, `saved`, `proposed`, `published`, `accessed`, and `frozen` are not only intended to say something about the relative quality of a particular object version, but also reflect whether an object version is in a developer's private experimentation domain, or in the project's official domain, where it is visible and accessible to the entirety of a project team. A more detailed description of the version control system can be found in [Mahl88a].

The first implementation of shape resembles Make. As in Make, the configuration process is controlled by a system description document. Unlike Make, however, shape operates on objects in an object base rather than UNIX filesystem objects. When building a certain configuration, shape searches the object base for appropriate versions of a system's component objects, installs them temporarily as UNIX filesystem objects (regular files), eventually invokes some of standard tools on them, and stores the resulting *derived objects* in the object base. Accordingly, the most significant improvement with respect to Make is the introduction of *selection rules* into the description document, the *Shapefile*. Selection rules control the selection process for component versions during identification, build or rebuild of system configurations.

The system description document used by shape consists of four basic components:

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

While the syntax for the system description part is the same as for Makefiles, the remaining shape-file 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.).

2.1. Selection of Component Objects

Configuration selection rules are used to *bind concrete object instances* in the object base to *component names* applied in the Shapefile at build time. A selection rule is a named sequence of *alternatives*, separated by semicolons, constituting a logical OR expression. Each alternative consists of a sequence of *predicates*, separated by commas, constituting a logical AND expression. A selection rule *succeeds* if one of its alternatives succeeds (i.e. leads to the unique identification of some object 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, andy),
        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, andy),
        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 attribute of a sought component is matched. The patterns used by shape have the same structure as those used by `ed`. The other predicates allow certain requirements to be expressed with respect to the object attributes. *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 lexical order) a given value. The similarly predefined predicate *attrmax* requires objects to have a maximal value in the specified attribute. In order to uniquely identify exactly one object 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 guarantees a unique selection. Predicates like *attrmax (attribute_name), attr (state, busy),* or *attr (version, 4.2)* are examples of such selections.

The basic procedure of shape is similar to Make's: Names – so-called *targets* – are produced. 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 objects nor the derivation context have changed. 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. During rule evaluation, the name is sequentially matched against the name pattern of the rule alternatives. If a pattern matches a name, the attempt will be made to complete the following predicate sequence.

After a name match, shape queries the object base for all objects with the given name and thereby initializes 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
- upon completion of the alternative, the cardinality of the hit set is not equal to one, i.e. no unique name resolution was possible.

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

The application of predicates to the hit set results in the removal of all AFS objects from the hit set that do not conform to the specified requirements. 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 transformation predicate is applied to the target and each of its dependents.

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. testsystems 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 object in the current directory*"), is active.

After having uniquely identified the system's components, they will be temporarily installed as UNIX filesystem objects, and the necessary transformation actions can be performed.

The search space for retrieve operations is defined by the *current project* which is an explicitly selected work context. If shape cannot find a document in the local search space, it tries to connect to a *project server*, whose address is 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.

2.2. Transformation Rules

shape's transformation rules describe, which kind of *source object* is transformed into which kind of *derived object*, how this transformation is performed, and which parameters are significant for that transformation. Transformation rules consist of a *transformation specification*, and a *transformation script*. Transformation specifications describe prerequisites and resulting objects of a transformation. The transformation script is a template for a shell script that will be passed to a shell process upon activation of a transformation. The syntax for these rules is an extension of Make's default rule specification formalism. shape's rule by which the transformation of a C source object into a '.o' (speak: dot-o) derived object is defined looks like:

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

A request to produce a derived object – let's say `xyzzy.o` – would, by evaluating the transformation rule, result in the compilation of `xyzzy.c` with the compiler defined by the macro `CC`, with compile switches defined by `CFLAGS`. The percent sign is consistently substituted by the name "xyzzy" throughout the transformation rule. The notation *$(NAME)* substitutes the value of the macro *NAME*, while *+(NAME)* substitutes the entire macro definition (i.e. *NAME=value*).

When a transformation rule is evaluated, the name of the source object is passed to the current selection rule and thereby bound to a concrete source object version in the object base. After the transformation script has been executed (successfully), shape assumes that the specified derived object has been produced, and stores the filesystem object that represents the derived object (in the example above `xyzzy.o`) in a special area in the object base, called the *binary pool*. When put into the binary pool, an attribute describing the transformation that just took place, is attached to the derived object. This *derivation attribute* contains the identification of all source objects, as well as the definitions of the specified macros that were used in the transformation script.

In Make, triggering of transformations is controlled by the built-in predicate *is_newer (target, source)*. This predicate fails to produce a target, if a source object is left unchanged, but something in the production environment changed. A common example of this case in UNIX is the compilation of the same C source file with different precompiler switches. On the other hand, transformations can be omitted, if the source was a saved version and the derived object is still existent in the binary pool. When requested to produce a target, shape configures the source objects (by evaluating the selection rule), constructs the derivation attribute that *would be* assigned to the derived object, and checks if a derived object that already has such an attribute is readily available from the binary pool. In this case no compilation would be necessary. If something in the source context or the transformation context has changed (e.g. the compile switches in `CFLAGS`), a new derived object instance would be created.

2.3. Variant Support

There is a common understanding of variants as *alternative* realizations of the same concept. 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.

Variant documents can be represented by a single source document that has to be *preprocessed* in order to access a particular variant. Variant documents can also be *physically separated*, i.e. each variant is a document of its own. Variants of this kind may have different names, or the same name but in different name spaces (e.g. directories). Variants can be identified by supplying a set of options to the preprocessor or appropriate names or pathnames. Both techniques have their particular advantages and disadvantages. In the case of preprocessed variants, one has to write source documents that contain lots of precompiler- or macroprocessor-related code. When things grow complex, it is easy to make mistakes. Besides, source documents of this kind are hard to read and understand. Consequently, they are less maintainable. Also, the specification of a planned configuration may require a lot of error-prone work.

The other case, the physically separate administration of variants, is expensive in terms of disk space and organizational overhead. Moreover, it is extremely problematic when changes (e.g. bug-fixes, feature enhancements) are made in sections that are common to all variants. One has to face the choice of

- doing the work all over again for all variants,
- trying to *merge* changes that have been made in one variant (semi-) automatically into the others, or
- changing only one variant (the best selling) and abandoning the others.

The real problem is even more complex. We suspect that it is still an unsolved problem of software engineering to produce *portable software designs* in the sense of predicting and planning the possibility that certain modules of a system sprout variant branches. It is still a fact that variants *happen*. In order to port a system to some new architecture (that might not have even existed at the time of the system's design), almost any module might be struck by the necessity of minor or major adjustments. As a consequence, both techniques of variant administration may happen to be mixed. Also variants of variants (sub- or multivariants) may come into existence. A complex system often has lots of variant options that are completely independent (e.g. a compiler that produces French diagnostics and VAX code for a UNIX System V environment). Variant properties of different modules might also be dependent or contradictory (e.g. certain memory management algorithms may rely on virtual memory, which a specified target architecture lacks). Furthermore, it is easily possible that the overall structure of a software system becomes variant. All this comes to a climax at a point where module variation is a mere symptom of metamorphosis, in which a system variant evolves into a completely different product. The question arises, how long variants should be treated as such, and when they start a life of their own.

We don't claim that shape offers a "quick fix" for the variant problem. For tools like shape, only flexibility, rather than conceptual or methodological corsets can be an answer. shape offers a simple but flexible mechanism for the handling of variants that supports both elementary administration concepts and their combination. For the realization in a UNIX filesystem environment, shape allows *variant names* to be defined and to be associated with variant-flags, that may be passed to the transforming tool, and a variant-path, that extends the search space by directories, which could host an object variant. The following example illustrates the use of variant definitions:

```
#% VARIANT-SECTION
vclass system ::= (vaxbsd, munix)
vclass database ::= (damokles, unixfs)
vaxbsd:
     vflags="-DUNIX -DBSD43 -DSTDCC \
        -DVAX -DPSDEBUG"
     vpath="sys/vaxbsd"
unixfs:
     vflags="-DFS"
     vpath="data/unixfs"
     ( . . . )
#% END-VARIANT-SECTION
```

Variant names, as defined in the variant section, can be applied in selection rules through the predefined (pseudo-)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 the variant attribute might look like:

```
fsexp:
  *.[ch], attrvar (vaxbsd), \
    attrvar (unixfs), 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 rule `fsexp`, *vflags* would become:

```
vflags="-DUNIX -DBSD43 -DSTDCC \
     -DVAX -DPSDEBUG -DFS"
```

and *vpath* would be:

```
vpath="sys/vaxbsd: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. 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. For a more thorough discussion of this matter, we refer to [Perr87a].

2.4. Configuration Identification and Reproducibility

shape can, for the purpose of configuration identification, be asked to produce a *configuration identification document* (CID) for a configuration. CIDs have the form of a Shapefile that is *completely bound*. All components of the configuration are precisely identified by their version number, and eventually variant identification. The CID also contains all necessary macro definitions. It is a good idea to place the tools that are used to create system configurations under version control as well. If this has been done, the tool versions are also included in the CID. Another important information recorded in the CID is the *project domain*. It defines the name of the project from which a configuration originates, and the next-higher domain in which the projectname can be resolved (e.g. a network–, host–, or pathname). This is necessary in order to define the object base against which a possible rebuild of the configuration shall be done.

Recording of system configurations in CIDs is the prerequisite for proper rebuilds of a complex system. The ability to rebuild a released configuration is in turn the prerequisite for proper product maintenance. The fact that shape CIDs are themselves Shapefiles makes it very easy to reproduce a recorded configuration, provided the original object base is available and intact.

However, complete reproducibility of system releases is a goal that is hard to achieve. To rebuild old configurations as identically as possible not only requires keeping 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 *tool registry*, where all production tools are logged. The gain on the other hand is increased security for the maintenance phase of long lived software products. The shape toolkit will offer support for tool registries but not require to keep them.

3. The Object Base

Most conventional filesystems, such as those of the UNIX flavor, solely use a file's (path-) name to establish access to its contents. They usually offer no provision for the maintenance of object histories. Version control systems like SCCS or RCS had to be introduced as auxiliary tools to avoid the problem of conflicting filenames and to store versions in an efficient manner. Histories of related objects are typically maintained completely independently, so there is no obvious way to figure out which versions of which objects were meant for each other. Integration of version control with system building tools like Make is very poor. The choice, which component versions go into a system configuration, is often limited to the selection of either the *newest available* versions or *statically designated*, i.e. a priori known versions (composition list).

For configuration identification it is more desirable to *describe* a planned configuration in terms of properties that its components must have in order to be eligible. The *attributed filesystem* (AFS) provides an extended view of software objects to application programs. This view comprises the concept of *object histories*, as well as the possibility to tag any number of *user-defined attributes* to object instances. Interpretation and use of these attributes is left to the application that uses them.

3.1. The Attributed Filesystem

The term attributed filesystem is only used for historical reasons. The AFS layer described in this section rather implements a data storage system independent base of *attributed software objects*† (ASO), on top of which all programs of the configuration engineering toolkit are built.

Each software object is understood as a complex of *content data* and a set of *associated attributes*. Attributes have the general form *name=value*, where *name* represents the attribute name, and *value* a possibly empty attribute value. There are some implicitly defined *standard attributes* for every object instance. Some of the standard attributes (e.g. name, size, owner, or protection attributes) are inherited from the underlying data storage system; others (e.g. revision number or state) are AFS specific.

AFS is realized as a callable interface to a *data storage system* and provides a general, attribute based method to identify objects in the object base. All attributes are considered equally suited for object identification. For instance, it is possible to retrieve all ASOs with an attribute *name=xyzzy*, or all items that have the attributes *author=andy* and *state=published*. Versions are treated as "just another attribute". AFS provides applications – such as shape – with fully transparent, stream oriented access to all object instances.

†We adhere to the terminology proposed in [Tich88a].

The AFS and its applications

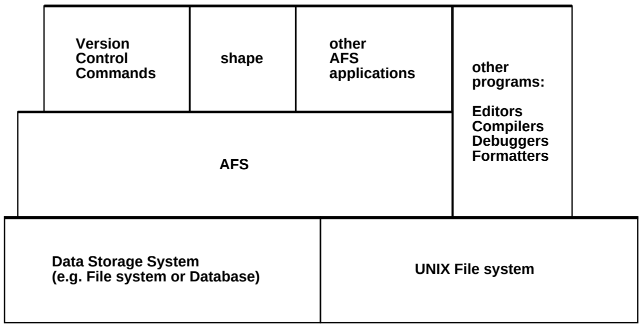

Fig. 3.1. The AFS and its applications

With AFS, software objects are retrieved by specifying an *attribute pattern*. An AFS retrieval results in a – possibly empty – set of *object keys*, each of which representing a unique object instance that matches the specified attribute pattern.

The current implementation of AFS uses two kinds of special files to realize the described concept of ASOs, *history files* and *binary pools*. Both file types are archives and capable of holding a possibly large number of objects. History files store successive versions of source objects, which are in most cases of textual nature. To preserve diskspace, only differences between successive versions are kept (*delta technique*). The version control system employs a new delta technique [Obst87a], yielding deltas that are about 30 percent smaller than deltas generated by *diff* (the standard UNIX comparison tool for generating deltas).

Binary pools are designed to hold *derived* objects, i.e. objects that are produced by processing (transforming, compiling) of source objects. Since derived objects can be reproduced from a corresponding source object, binary pools are administered in a cache fashion, i.e. when space runs short, binaries are cleaned out on a "least accessed" basis. The concept of binary pools resembles DSEE's *derived object pools*.

Similar to other version control systems, history files generally accompany a regular file, the so called workfile. In the context of shape and AFS, we use the term *busy object*, instead of workfile. Busy objects are regular files, thus making it possible to use tools, such as the EMACS editor, which don't know about the AFS. Access to non busy objects however, can only be established through the AFS interface, because usually they must first be regenerated from the information stored in the corresponding history file. When a saved version is opened, the I/O operations will actually access a temporarily created file. All the action happens – of course – behind the scenes, and an AFS application accesses busy objects as well as all other versions transparently through the same mechanism.

The attributes of all version instances of an object, including the busy object, are stored in the history file. Attributes of derived objects are stored in the binary pool file that hosts them.

Every regular file can be accessed through the AFS. In this case, it will be treated as a busy object without AFS specific or user-defined attributes. If such a file is checked into the version control system for instance, a history file will be created, and the missing standard attributes will be supplemented in a meaningful manner.

3.2. AFS on top of DAMOKLES

A second implementation of AFS on top of DAMOKLES, a dedicated software engineering database (prototype) is currently under way. DAMOKLES is based on an *extended entity relationship model* and designed to serve as universal data management system for a large set of tools to be part of an integrated programming environment.

The fact that the various tools are supposed to store their data in a tool specific structure (of entities and relationships) leads to some serious problems. First, opening of an ASO containing application specific DAMOKLES structures must yield an object specific key rather than a UNIX file pointer. Second, simple operations such as copying, moving, or printing of object contents are representation dependent and can't be performed by the AFS any more. In particular, there is no easy way for AFS to compute *version deltas* between DAMOKLES object structures (how should this kind of delta look like ?).

Another very practical problem is extensibility. In the current state, a DAMOKLES database has *one* schema that defines the internal structure of the entire database. Addition of a new tool with the necessity of schema updates requires the generation of a new database and the (troublesome) transfer of all data from the old database to the new one. Due to these problems and the current lack of of DAMOKLES based tools, we decided to use the database only as replacement for archive– and binary pool files. Busy objects and temporary files (containing the data of reconstructed older versions) are still stored in the UNIX filesystem. AFS applications have no direct access to the database. Similar to the filesystem implementation, the contents of an ASO has the form of a byte array. Until now, there is no substantial difference between this implementation and the filesystem implementation — except that it will be slower.

A solution to the described problems that will allow AFS applications to take full advantage of DAMOKLES' structuring facilities while supporting the dynamic extension of the SPU without the necessity to change or even recompile any part of the environment, can be found by taking a consequently object oriented approach. The design of an object oriented AFS to be basic part of an object oriented *environment frame* is currently in its early stage. However, we expect substantial results in this direction in the near future.

4. Conclusion

The described system for engineering of software configurations integrates tools for version control, system building, and project organization on top of a common object concept. Despite the integration of the toolkit itself, care has been taken to smoothly fit the SCE system into the standard UNIX environment. The entry cost for new users (in terms of initial learning effort) is very low. The system may be used as just a friendly version control tool, an upward compatible replacement for Make, or an intelligent integration of both. Utilization of the toolkit can be gradually extended up to a network based project support environment, employing a number of programmer teams working on large software development and maintenance projects.

The simple, nature of the toolkit's basic user interface offers a broad range of possibilities to create dedicated user interfaces that provide an integrated, problem oriented view of the SCE system.

There are, however, a number of areas where much more work needs to be done. Particularly, powerful formalisms to deal with the complexity of variant control are still at large. The use of software engineering datbases as underlying data storage system is still a problem. However, the use of object oriented design principles for a new version of the system is expected to overcome these problems and to open up exciting perspectives for a "generic, integrated, open software engineering toolkit environment".

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

AUGM83a.
AUGMAKE, "An Augmented Version of Make," *System V/68 Support Tools Guide*, pp. 21-40, Motorola Microsystems Inc., Ord.No. M68KUNSTG/D1, January 1983.

Bers80a.
Edward H. Bersoff, Vilas D. Henderson, and Stanley G. Siegel, *Software Configuration Management,* Prentice Hall, Englewood Cliffs, N.J., 1980.

Clem86a.
Geoffrey M. Clemm, "The Odin System: An Object Manager for Extensible Software Environments," *CU-CS-314-86*, The University of Colorado, Boulder, Colorado, February 1986.

DECC84a.
DECCMS, "VAX DEC/CMS 2.0 Reference Manual," *VAX DEC/CMS Reference Manual, Order No. AA-L372B-TE*, Digital Equipment Corporation, Maynard, Massachusetts, November 1984.

DECM84a.
DECMMS, "VAX DEC/MMS 2.0 User's Guide," *VAX DEC/MMS User's Guide, Order No. AA-P119B-TE*, Digital Equipment Corporation, Maynard, Massachusetts, August 1984.

Ditt86a.
Klaus Dittrich, Willi Gotthard, and Peter C. Lockemann, "DAMOKLES. A Database System for Software Engineering Environments," *International Workshop on Advanced Programming Environments*, pp. 351-371, Springer Verlag, Trondheim, Norway, June 1986.

Dows86a.
Mark Dowson, "ISTAR - An Integrated Project Support Environment," *SIGPLAN Notices*, vol.

22, 1, pp. 16-27, ACM, Palo Alto, California, December 1986.

Estu86a.
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.

Feld79a.
Stuart I. Feldman, "MAKE - A Program for Maintaining Computer Programs," *Software - Practice and Experience*, vol. 9,3, pp. 255-265, March 1979.

Habe82a.
A. Nico Haberman and Gail E. Kaiser, "An Environment for System Version Control," *Dig. Papers Spring Compcon*, IEEE, November 1982.

H87a.W. Hbner, G. Lux-Mlders, and M. Muth, "THESEUS – Die Benutzungsoberflche der UNIBASE Softwareentwicklungsumgebung," *Beitrge zur Graphischen Datenverarbeitung*, Springer Verlag, Berlin, 1987.

IEEE83a.
IEEE, *IEEE Standard Glossary for Software Engineering Terminology,* IEEE, New York, N.Y., February 1983.

Lamp88a.
Andreas Lampen and Axel Mahler, "An Object Base for Attributed Software Objects," *(to appear in) Proceedings of the Fall 1988 EUUG Conference*, European Unix systems User Group, Lisbon, Portugal, October 1988.

Lamp83a.
B. Lampson and E. Schmidt, "Organizing Software in a Distributed Environment," *Proc. of the SIGPLAN 83 Symposium on Programming Language Issues in Software Systems*, vol. 18, 6, pp. 1-13, San Francisco, California, June 1983.

Lebl84a.
David B. Leblang and Robert P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices*, vol. 19, 5, pp. 104-113, ACM, Pittsburgh, PA., April 1984.

Mahl88a.
Axel Mahler and Andreas Lampen, "shape – A Software Configuration Management Tool," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 228-243, B.G. Teubner, Grassau, West-Germany, January 1988.

Marz86a.
Keith Marzullo and Douglas Wiebe, "Jasmine: A Software System Modelling Facility," *SIGPLAN Notices*, vol. 22, 1, pp. 121-130, ACM, Palo Alto, California, December 1986.

Nick83a.
Peter J. Nicklin, *The SPMS Software Project Management System,* UC Berkeley, Berkeley, CA., August 1983.

Obst87a.
Wolfgang Obst, "Delta Technique and String-to-String Correction," *Lecture Notes in Computer Science*, pp. 69-73, Springer Verlag, Berlin, September 1987.

Perr87a.
Dewayne E. Perry, "Version Control in the Inscape Environment," *Proceedings of the 9th International Conference on Software Engineering*, pp. 142-149, IEEE, Monterey, California, March 1987.

Roch75a.
Marc J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 364-370, 1975.

Tich85a.
Walter F. Tichy, "RCS - A System for Version Control," *Software - Practice and Experience*, vol. 15,7, pp. 637-654, July 1985.

Tich88a.
Walter F. Tichy, "Tools for Software Configuration Management," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1-20, German Chapter of the ACM, Grassau, FRG, January 1988.

Wait88a.
William M. Waite, V. P. Heuring, and Uwe Kastens, "Configuration Control in Compiler Construction," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 228-243, B.G. Teubner, Grassau, West-Germany, January 1988.