An Object Base for Attributed Software Objects†

Andreas Lampen, Axel Mahler
andy@coma.uucp, axel@coma.uucp

Technische Universität Berlin
Sekretariat FR 5-6
Franklinstr. 28/29
D-1000 Berlin 10

ABSTRACT

The UNIX filesystem supports a fixed set of attributes for filesystem objects, stored in inodes and directory entries. The (path-)name attribute is the sole means to identify and access a filesystem object. This turns out to be a rather severe limitation for certain complex applications such as large scale software development, where software objects typically evolve in a considerable number of versions.

An approach to improve the situation is introduced by the *attribute filesystem* (AFS), the system described in this paper. The AFS combines the notion of immutable objects (versions) with the possibility to attach any number of user-definable attributes to any attributed software object (ASO). AFS objects can be identified by specifying any set of attribute value(s) as retrieve pattern. The *name* of an AFS object is treated as *just another attribute*. The AFS is equipped with a proper retrieve interface that allows non-unique identification of *sets of objects* and provides operations on those sets.

The AFS is a significant extension to the UNIX filesystem interface providing applications with a unified, consistent view of attributed filesystem objects comprising immutable versions and ordinary UNIX files. The concept of *persistent objects* makes AFS a basis for object oriented applications in an UNIX environment. We used AFS as a basis for the implementation of the **shape** toolkit, a collection of programs supporting software configuration management in multi-programmer software development projects.

One important objective of AFS is to abstract from the actually underlying data storage system. This paper will briefly discuss two different implementations of AFS − one on top of the UNIX filesystem and the other based on **DAMOKLES**, a dedicated software engineering database.

1. Introduction

The context of this work was the design of a software configuration management system to be part of a kernel of an open UNIX based software engineering environment. Large scale software development requires a system capable to store a big number of software objects, typically evolving in a considerable number of versions.

The UNIX filesystem is a good basis for this job but it has some severe limitations. It supports only a fixed set of attributes for filesystem objects, stored in inodes and directory entries. The (path-)name attribute is the sole means to identify and access a filesystem object. Version control systems had to be introduced as auxiliary tools, to avoid name clashes and to store multiple versions of files in an space efficient manner. However, they do not overcome the fundamental shortcomings of the UNIX filesystem. They represent

† in *Proceedings of the EUUG Autumn '88 Conference*
(Cascais, 3-7 October 1988), pp. 95-105, European UNIX User Group, London 1988

partial solutions to particular problems and integrate poorly with UNIX tools such as Make [2].

As a try to overcome these limitations we designed the *attribute filesystem* (AFS), the system described in this paper. We use the name attribute *filesystem* mainly for historical reasons (because we are used to it and we haven't found a better name yet). So don't be confused when we subsequently refer to AFS as an *object base* rather than a filesystem. Accordingly we will use the term *software object* instead of *file* throughout this paper.

An object base is a hybrid of filesystem and database trying to combine the advantages of both types of data storage systems. The advantages of an object base over a filesystem lie in its

- more general object identification scheme,
- ability to store multiple versions of objects,
- support for the management of arbitrary attributes for objects and
- abstraction from the physical representation of objects.

Improvements in respect to database systems are

- the concept *object* as manageable, user identifyable, interchangeable piece of data and a common understanding of how data looks like (a basis for integration of existing and new tools),
- no need for a schema (i.e. no problems with schema updates),
- better network support (*objects* are quite easyly interchangeable between different machines) and
- better performance (in most cases)

For a detailed discussion on how a persistent object base should look like, see [6].

AFS is designed as basis for the **shape**-toolkit, described in [4] and [5]. The **shape**-toolkit is a configuration management toolkit integrating a dedicated version control system and **shape**, a significantly enhanced Make program. Enhancements include full access to the version control system. **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 next chapter gives an overview of the basic concepts of AFS. In chapter 3, we briefly describe the interface and have a look at two different implementations. The last chapter contains an outlook on current development efforts to make AFS a basis for object oriented applications.

2. General design

AFS is a system to store software objects as a complex of (from the AFS' point of view) unstructured content data and an arbitrary number of associated attributes. We call these objects *attributed software objects* (ASOs). AFS has a built-in version control system that supports the storage and identification of different versions of *design objects*. A design object consists of a group of successive versions evolving by taking snapshots of the current development state from time to time. Throughout this paper we also use the term *line of development* synonymously to *design object*. Each version of a design object is an attributed software object on its own. For a more detailed discussion of software engineering terms see [10].

Due to the requirements of a multi programmer software development environment we introduced a mechanism for reserving update rights for design objects by setting a lock on this object. Reserving the update right has the effect, that updates can only be stored by the user currently holding the lock.

AFS was designed to be a basis for integration of a large spectrum of different software development tools. These may be editors, compilers, formatters, a software configuration management system, tools for project planning and status accounting, test support, specification and design tools — a potentially very long list. In respect to tool integration, an engineering environment has to offer some form of intelligent support for the user in order to improve the situation presented by mere toolbox systems. However, it is not feasible for the object management part of such an environment to understand the formal semantics contained *in* the various types of objects it manages. The functional behavior of an environment's object manager should rather be guided by simple information *about* the managed objects. Attributes enable each AFS application program to store information about a software object persistently either for later use or leaving them for interpretation by other AFS applications (or both).

It is likely that the number of needed attributes will grow with each new tool using AFS. As we cannot foresee which new AFS applications will be introduced in the future, we are not able to define a fixed set of attributes serving all needs. Instead, we decided to keep the number of attributes for each stored object dynamically extensible. Besides some standard attributes (such as name, size or version), each software object is stored with a potentially infinite number of application-defined attributes.

Another important design goal was the introduction of a concept for the integration of standard UNIX tools and off–the–shelf tools that operate on regular files. For this purpose, AFS is designed to be able to coexist with the UNIX filesystem. It allows regular UNIX files to be treated as AFS objects and to be accessed through the AFS interface. This opens a way to exchange data between existing tools that operate on UNIX files and tools on top of AFS (such as the **shape** toolkit). Figure 2.1 illustrates the tool integration on base of AFS. AFS is a C–language interface that abstracts from the underlying data storage system (e.g. a filesystem or a database).

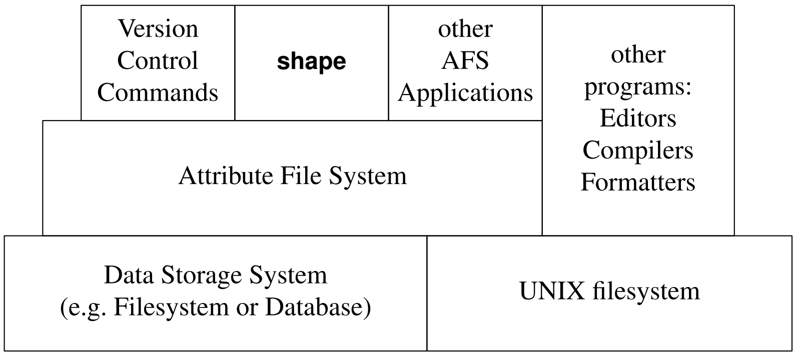

Fig. 2.1: The AFS and its applications

Apart from versions and attributes, AFS looks very much like the UNIX filesystem with a hierarchical directory structure, special files and so on.

2.1. Source objects and derived objects

AFS manages *lines of development*. A line of development consists of a *busy object* and a number of saved versions. The busy object is an ordinary alterable UNIX file. It can be accessed via AFS *and* UNIX filesytem operations. Versions come into being by making a copy of the current state of the busy object. They can be stored as source objects or as derived objects.

Source objects are typically composed manually (e.g. by use of a text editor). AFS stores source objects as immutable objects. Once saved, they cannot be modified any longer. They can only be explicitly removed by their owner or author. Source objects can be uniquely identified by the attribute values for *hostname, system path, name, type, generation number, revision number* and *variant*.

Derived objects are typically derived automatically (e.g. by a compiler) from a source object and thus be reproducible at any time. They are kept in a *derived object pool* (see also [3]), a data store of limited size that is administered in a cache fashion. The oldest (access date) derived objects get removed if space runs short in the binary pool. Unlike source objects, the tupel (*hostname, system path, name, type, generation number, revision number, variant*) does not necessarily have to be unique for derived objects. Derived objects may differ only in user defined attributes !

AFS makes no assumptions whether a copy of a busy object shall be stored as source object or as derived object. This has to be decided by the application.

2.2. Attributed Software Objects

Attributes have the general form *name=[value [value ...]]*. Figure 2.2 illustrates the term attributed software object.

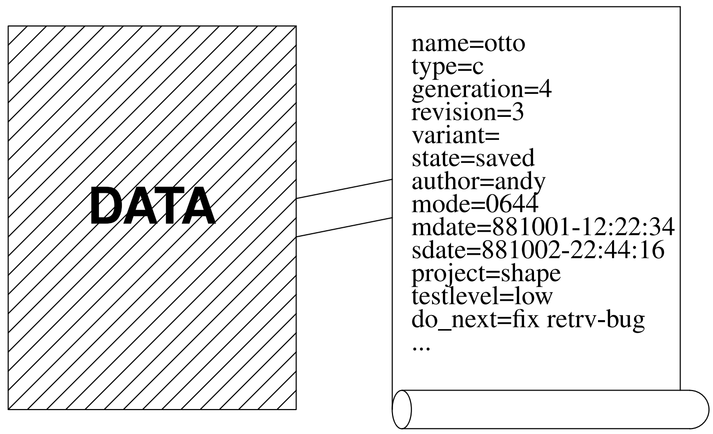

Fig. 2.2: An Attributed software object

AFS supports a set of standard attributes for each ASO. These are

- hostname, system path, name, type,
- generation number, revision number, variant name, version state,
- owner, author,
- size,
- access permissions, lock,
- date of last modification, – last access, – last status change, – saving and – last lock change

The *system path* attribute is used to describe the (logical) position of an ASO in a complex, hierarchically structured system. *System path, name* and *type* are currently mapped to UNIX pathnames where they take the place of path prefix, filename (without suffix) and filename suffix. For example: the system path `/usr/axel`, name `foo` and type `c` are mapped to the UNIX filename `/usr/axel/foo.c`. Additionally to the *owner* attribute known from UNIX files ASOs also have an *author* attribute. AFS distinguishes the *owner* of an entire design object and the *author* of a particular version.

In addition to the standard attributes, an application may attach any number of *user defined attributes* to any ASO. Interpretation and use of these attributes is left to the application that uses them. AFS manages user defined attributes as a structure containing the attribute name and a (possibly empty) set of strings holding the associated attribute value(s).

Attributes can be used for very different purposes. The standard attributes are used mainly for ordering and identifying software objects and for general management. User defined attributes will be used to store application specific information with an ASO.

A typical example for the use of user defined attributes can be found in the **shape** program. For each derived object (e.g. an object file that is produced by compiling a C source file), the name of the transformation tool and all given flags and parameters are protocoled in an attribute. Besides the description of the transformation process, the name(s) and version(s) of the involved source object(s) are recorded as list of values in another attribute. This enables **shape** to handle recompilations in a much more sophisticated way than Make.

Another example might be a tiny program that allows a user to associate any number of keywords with his software objects. Such a program would allow a grouping of ASOs via keywords independently of the directory structure. User defined attributes can also be used to model references between ASOs. Attaching the identification of the corresponding source object as attribute to a derived object actually *is* a reference.

The full power of the concept of user defined attributes is hardly used yet but we feel that there *is* a lot of power in this approach. Our first application programs use this mechanism in a straightforward manner (illustrated in the examples above) and it works fine. User defined attributes may also be used in a more exotic way such as to store patches (e.g. ed scripts generated by *diff*) to source objects.

On the basis of user defined attributes it will further be quite easy to construct a general object class manager that provides an interface for the storage of typed software objects with type specific attributes. This will be discussed in greater detail in chapter 4.

2.3. Identification and Object-Keys

Due to the introduction of versions, ASOs cannot be uniquely identified by just giving their (path-)name (like in the UNIX filesystem). For the identification of attributed software objects, all attributes are considered equally qualified. AFS is equipped with a retrieve interface for nonunique identification of sets of ASOs. A retrieve operation is driven by an attribute pattern containing all desired attribute values.

Once identified, an ASO is represented by a unique *Object-key* by which it can be accessed further on. Object keys can be fetched from a set, delivered by a retrieve operation. Selecting an object key from a set is supported by operations for sorting sets by attribute values, recursive retrieve (a retrieve operation with a set as search space) and building unions, intersections, or differences between two sets.

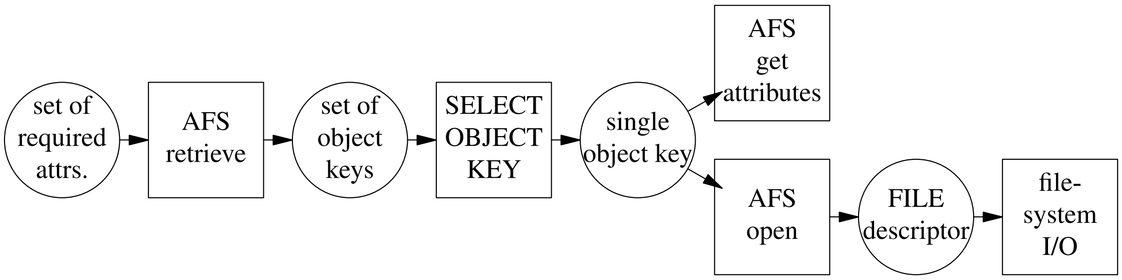

Fig. 2.3: Access principle for ASOs

Access to the attributes as well as the contents on an ASO is also established via object keys. To access its contents, an ASO has to be opened. Once opened, the content data of an ASO can be read and written as regular UNIX files (see fig. 2.3).

2.4. Version Control

AFS maintains a sequence of *revisions* (successive versions) for each design object. The numbering of revisions in a line of development is under control of the AFS. Every time when a new revision is generated by saving a copy of the busy object, it is assigned a version identification of the form *generation.revision* (e.g. 1.0 or 4.3). Generation numbers indicate major development steps that are common to all components of a complex system. The revision number serves as an update sequence number for individual components within a generation.

Besides revisions, a version control system has to be able to store *variants* – qualitatively equivalent versions differing in a certain property – of source objects. A typical example for variants is a user interface module that deals with different window systems. Unlike the identification of revisions, where a generation.revision numbering scheme turned out to be common sense, there is no usual variant naming scheme. We know systems, where variants are numbered, named (with single or multiple names) or both.

The general management of variants is a major conceptual problem. The idea of variants sounds quite simple, but to actually handle them can be an extremely difficult task. Up till now there doesn't seem to be a tool that conducts the development of system variants in a really satisfying manner. Nevertheless we want to establish a reasonable basis for such a tool by providing an instructive and flexible variant identification mechanism. AFS supports a *variant* attribute (a string) for each ASO. The variant attribute is totally independent from the revision numbering, i.e. each variant is stored in its own line of development with independent revision numbering. Any structure of the variant string (e.g. multiple variant names) has to be maintained by the application(s) [5].

2.4.1. Branches

One major difference between the AFS builtin version control system and classical systems like RCS [9] and SCCS [8] is, that AFS establishes a user oriented view of versions that abstracts from the physical derivation graph. In contrary to RCS and SCCS, version histories in AFS are essentially linear. Logically, new versions are always appended to the end of the version history. Updates of older versions can be attached as leaves (not branches) to the line of development (see fig 2.5). Variants are stored in parallel lines of development.

At first glance, it looks like a loss of functionality of the AFS version control system compared to systems like RCS or SCCS (where are the branches). A closer look shows, that there is only a different identification scheme.

Figures 2.4 and 2.5. show the same number of revisions/variants developed in the same way. The first figure shows a RCS version tree and the second shows its AFS pendant.

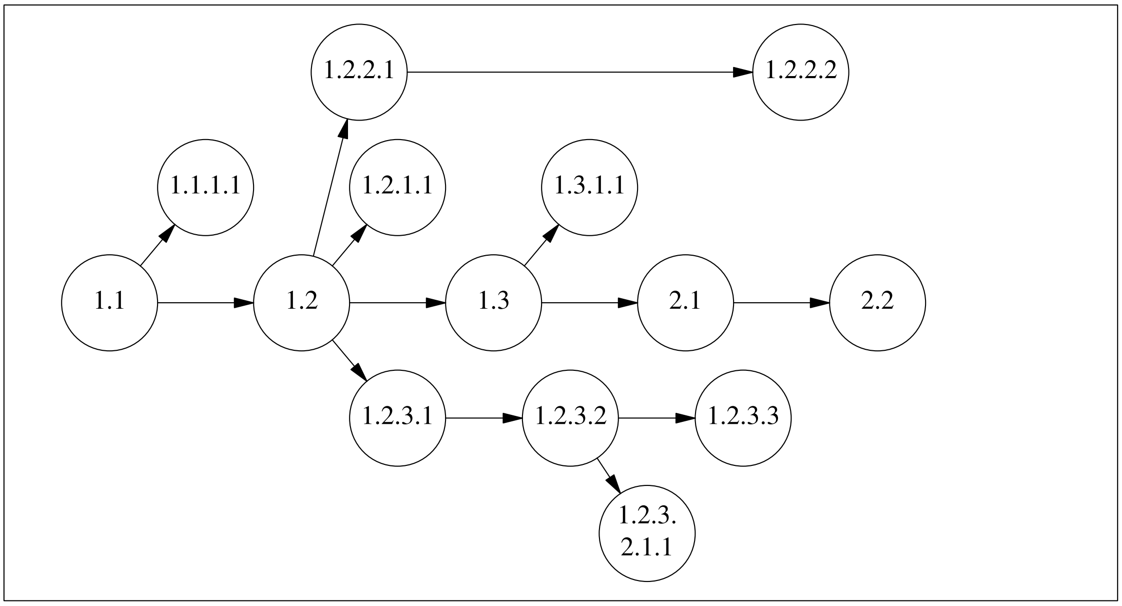

Fig. 2.4: RCS version tree

The version identification scheme of AFS abstracts from the physical evolution of the single versions (nevertheless, the physical links are there and maintained in a way similar to RCS). RCS and SCCS present a data structure oriented view of version histories to the user. Some nasty problems arise from mixing logical and physical derivation graph in respect to numbering/naming of revisions. First, version numbers in branches get very complex (four or more numbers) which is not very instructive. Second, although there are "so many numbers" there is no generation number in branches !

Conceptually, AFS does not support arbitrary branching. Branching off the main line of development happens most of the time because a developer wants to

- create a variant or
- store modifications of older versions (bug-fixes or customizations) or
- make experimental changes that should not necessarily go into the main line of development.

The first case should be realized by parallel lines of development. The latter two cases are not explicitly supported by AFS itself. A solution has to be given by an application program either based on an extended use of the *variant* attribute or on storing patches in user defined attributes.

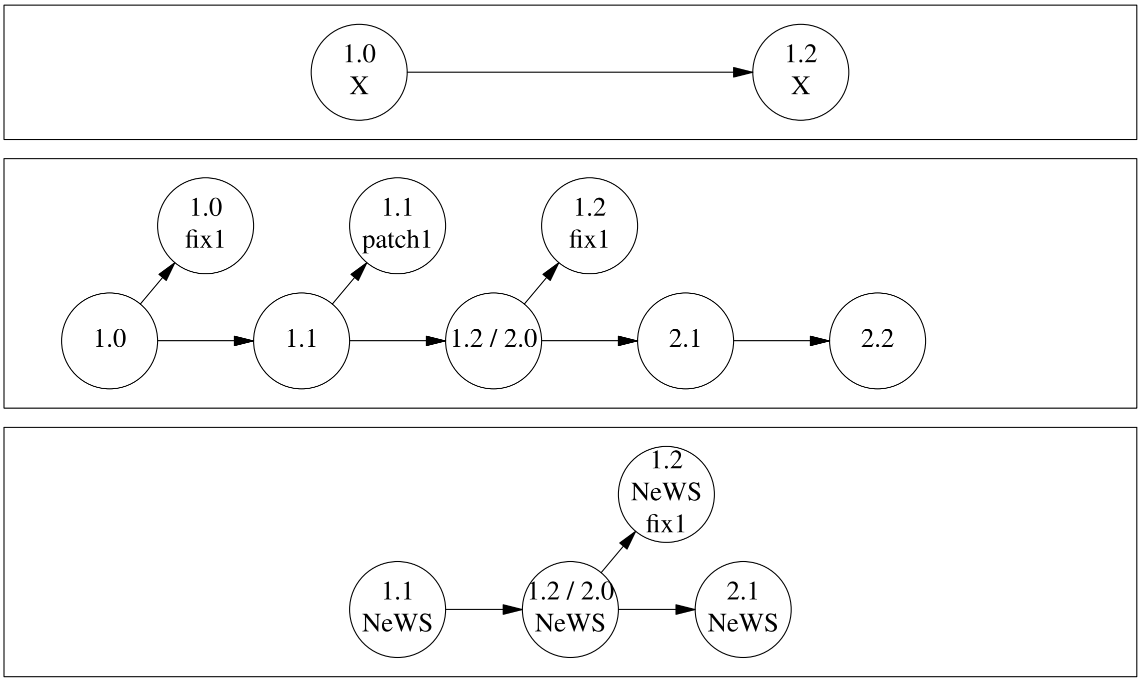

Fig. 2.5: AFS lines of development

2.4.2. The version status model

As a consequence of the original design of AFS as part of a software development environment, AFS provides a status model for saved source objects to support the implementation of a general project organization scheme. The project organization scheme we have in mind distinguishes between private workspaces for each participating programmer, and a central project database, where all the serious results of the programmers' efforts are collected.

The basic status model that is built into the version control system recognizes the following states:

busy The object is under development and its contents may be changed.

saved The object is saved and may be used for later backup.

proposed The object has been submitted for publication by the developer but still needs formal approval by a quality assurance board to become publically visible and accessible in the official database.

published The object has passed the formal approval and is now accessible to every member of the project. It is not yet accessed and may therefore be withdrawn if necessary.

accessed The object has been published, and is now accessed by some members of the project.

frozen The object may be part of a system configuration that has been released to the outside world. This means it must under no circumstances be destroyed.

3. Realization

Currently, we have two different implementations of AFS (in different development stages). One on top of the UNIX filesytem and another one on top of **DAMOKLES**, a dedicated software engineering database [1]. Both are written in C and provide a library interface. Further developments will be made in C++ and we will implement a new interface in C++.

The interface of both implementations is nearly identical. Differences exist only in marginal routines concerning the data management. For example has the **DAMOKLES** implementation additional routines for opening and closing the database.

3.1. The Interface

The main datatypes of AFS are:

- The *object key* that uniquely identifies a software object. The structure of this type is different in the two different implementations. Consequently, application programs should handle this type as opaque type and should not access single fields (unfortunately this is a little bit clumsy in C – an implementation as C++ class would be more appropriate).
- *Set descriptors* represent a set of object keys. A set descriptor contains information about the number of keys in the set and a pointer to a list of object keys.
- An *Attribute buffer* which is capable to hold all attributes of a software object (standard attributes and user defined attributes). Attribute buffers have two different purposes. First, they can hold a retrieve pattern, i.e. they may be (partially) filled with *desired* attribute values and then be passed as argument to a retrieve operation. Second, an attribute buffer is used to return all attributes of an identified ASO on demand.

The names of all operations of the AFS are preceded by *af_* (e.g. *af_saverev*). In the following sections, the most important (most interesting) functions of AFS are described.

3.1.1. Retrieval

AFS provides two different retrieval operations, *af_find* and *af_bpfind*. The first does retrieval of source objects and the second of derived objects. Both have to be parameterized with an attribute buffer containing a retrieve pattern and both return a set holding the keys of all ASOs matching the specified attribute pattern.

Due to the possibility to uniquely identify source objects by the attributes *hostname, syspath, name, type, generation, revision* and *variant*, there is a shortcut (faster) operation to get the key of well known object by specifying values for these seven attributes. The name of the function is *af_getkey* and it returns an object key if the sought software object exists.

3.1.2. Sets

AFS contains a considerable number of operations on sets. Although a set contains *object keys* and not complete objects, we speak of “objects in a set” throughout this chapter for reasons of simplicity.

Af_sortset sorts the objects in a set in respect to the value of a single or compound attribute. For example, objects can be ordered alphabetically by their name attribute which is a single attribute. A typical compound attribute is the *version number* consisting of generation number and revision number (generation.revision). Besides standard attributes one can pass any name of a user defined attribute as argument to *af_sortset*. This works, even if the attribute is not defined for every object in the set.

The call *af_subset* performs a recursive retrieve operation on an existing set. *Af_subset* is (like *af_find*) parameterized with an attribute buffer containing a retrieve pattern. It results in a subset of the original set.

Other set commands provide adding and deleting single objects to/from sets as well as building intersections, unions and differences of two existing sets.

3.1.3. Reading and Manipulating Attributes

The manipulation of standard attributes for software objects is under strict control of AFS. Many standard attributes can only be modified implicitly by performing a AFS operation. The other standard attributes can only be manipulated in a well defined manner.

The attributes *hostname, system path, name* and *type* are immutable for saved objects (source or derived). They are set once (at saving time) as inherited from the filesystem location of the file the object evolved from. As busy objects are regular UNIX files, they can be renamed by linking them to another place in the filesystem.

Version number and the version state are only relevant for source objects. A source object is automatically assigned a version number and state *saved* when it is generated. The version number can subsequently only be increased (set to a logically bigger number, e.g. from 1.2 to 1.4 or from 2.3 to 3.0). The version state can be set to the logically following or preceding (see states list in 2.4) state but it can never be reset to *busy*. Busy objects have no version number and are always in state *busy*. Version number and state of derived objects can be set arbitrarily (but AFS does not care about their values).

The management of user defined attributes is entirely controlled by the application program. An application program called by the owner or the author of an ASO can arbitrarily add, delete and modify these attributes. AFS supports the management of user defined attributes by storing them as name/value(s) pairs. A name (string) is associated with any number (zero to infinite) of values (strings). The function *af_sudattr* allows the addition and deletion of whole attribute entries and single values.

3.1.4. Locking

The locking mechanism for reserving update rights in a line of development is reflected by the functions *af_lock, af_unlock* and *af_testlock*. Calling the function *af_lock*, causes the identified ASO to be locked by the caller, provided the object was available for locking.

Af_testlock returns the user ID of the lock holder (if present). Locks can either be released by the lock holder of the design object or be broken by the owner by means of *af_unlock*.

3.1.5. Version Control

The version control system has been described in detail earlier in this paper. The most important operations are *af_saverev* and *af_savebinary* which create a copy of the given, busy, ASO and store it as new source resp. derived version.

Af_newgen increases the generation number in a line of development. It creates a pseudo revision which is identical to the last saved revision and attaches a new version number to it. The generated pseudo revision has the character of an alias (see also figure 2.5). The new version number is generated by increasing the generation number by 1 and resetting the revision number to 0 (e.g. 1.4 becomes 2.0).

If the busy object of a line of development was replaced by another file, for example an older version or a variant, *af_setbusy* should be called with the key of the old busy object and that of the new one, to inform AFS about the change. This is necessary to keep track of the physical derivation structure of a line of development which is substantially for an efficient delta technique.

3.1.6. Filesystem Primitives

Af_open and *af_close* are filesystem primitives analogous to the UNIX calls *fopen* and *fclose*. The only difference is, that *af_open* takes an object key instead of a pathname as input parameter. *Af_open* returns a pointer to a *FILE* structure, so that the application program can use the whole UNIX stdio interface on ASOs.

Additionally, ASOs can be created and removed (unlinked). Derived files residing in a binary pool can be restored at their former position with *af_restore*.

The interface is completed by a number of miscellaneous routines. These are routines for error reporting and cleaning up (removing tmp files and freeing memory) as well as functions for splitting up UNIX pathnames in the AFS attributes syspath, name and type.

3.2. AFS on top of the UNIX filesystem

The first implementation of AFS was done on top of the UNIX filesystem. This implementation is ready and in use.

Saved ASOs are stored in archive files (like SCCS and RCS). AFS maintains two archive files for each line of development of source objects – one holding the attributes and the other holding the data. AFS stores its archives in a subdirectory called AFS. If the archives shall be stored elsewhere but not in a subdirectory, an AFS application can set an *archive path* to specify the name of a directory where all subsequently accessed or created archives are stored. In this case, the application program is responsible for maintaining the relative–path relationship between a busy object and the corresponding archive.

To save disk space, Successive versions in a line of development are stored as deltas (differences to the previous version). For generating file deltas a new delta technique, described in [7] is employed. Deltas computed by this method are about 30 percent smaller than deltas generated by *diff* (the standard UNIX file comparison tool). This delta technique uses characters rather than lines as atomic symbols.

We use a backward delta technique (like RCS), i.e. only the most recent version of each line of development is stored completely. Older versions can be reconstructed by successively applying all deltas from the most recent to the desired version. In the filesystem implementation, deltas between parallel lines of development are not supported (conceptually, they should be there). This is because different lines of development are stored in different archive files and we consider it too risky to spread delta chains across different UNIX files. If the coherence of the two files gets lost (this may happen by simply moving one file) and the chain gets broken, some revisions may not be reproducible any more.

Derived files are stored physically unchanged. Tests with our delta technique showed, that it makes not much sense to build deltas between binary files. On calling *af_savebinary*, the identified file is linked into the AFS subdirectory. Additionally, the attribute buffer containing all the attributes for the derived file is stored in a *ndbm* Database.

If a saved source object shall be opened, it is reconstructed and stored in a temporary file. Due to the unalterability of saved ASOs, the temporary file is simply removed on closing (changes have no effect on the stored ASO). If one really wants to reconstruct a file, one has to create a busy object, read the data from the tmp file and write it to the created busy object.

3.3. AFS on top of DAMOKLES

The AFS implementation on top of **DAMOKLES**, is currently under development. In this implementation, saved revisions and binary pools are stored in the database.

In the current implementation, the database is only used for storing the internal data (archives and binary pools) of AFS. 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 stream. Up till now, there is no substantial difference of this impementation to the filesystem implementation except that it will be slower.

The actual enhancement of AFS on **DAMOKLES** will be the possibility for applications to use the structuring facilities of the database for their internal data (the content data of ASOs). But how shall a specific ASO be created, copied, or opened, and how should a delta between two versions be constructed ? Questions that can only be answered when the contents structure of the corresponding ASO is known. This problem can only be solved satisfactorily by following an object oriented approach. An AFS implementation taking this approach will provide a mechanism for the definition of ASO classes, defining specific attributes and virtual operations (e.g. compute delta, open, linearize) on instances of these classes.

Another very practical problem is extensibility. In the current state, a **DAMOKLES** database has *one* schema where all database object types have to be defined. Addition of a new application with the necessity of schema updates requires the generation of an entire new database and the (troublesome) transfer of all the data from the old database to the new one.

4. The future of AFS

When we made the first design of AFS, aimed primarily at facilitating the implementation of the **shape**–toolkit, we were not aware of the power of this approach. Especially the mechanism of user defined attributes inspired us to think of applications that go far beyond the tasks AFS was originally designed for.

However, it is still a problem to make consistent use of user defined attributes. Each application has to set them explicitly. The exchange of information via user defined attributes has to be subject of special agreements and naming conventions. A good improvement will be the introduction of some kind of ASO definition language. This would be the input for a general class manager that manages the storage of objects of well defined object classes. The definition of an object class contains a specification of all attributes attached to an object of this class. The object classes will be part of a class hierarchy with attribute inheritance. This makes AFS a basis for object–oriented software engineering applications that deal with software objects. These are able to store their objects persistently, identify them (uniquely) and share them with other applications.

Subsequently we want to give an idea, how a possible class hierarchy might look like. Examples for classes are *software object*, *source object*, *derived object* or *revisible object*. The class *software object* is the basic class. All other classes are derived classes inheriting the attributes of *software object*. *Source objects* and *derived objects* are direct subclasses of *software object*. *Revisible source object* would be a subclass of *source objects* with the additional attributes *generation number, revision number* and *version state*. An example for defining a class hierarchy for an object oriented database can be found in [11].

Another important topic is support for AFS applications distributed over a local area network. To support this, we need *persistent keys*. Unlike the current realization of object keys which are only valid during the lifetime of the application process, persistent keys are interchangeable between processes and machines.

5. Conclusion

Properly speaking, AFS was invented by accident. We needed a basic data storage system capable to hold the software objects we deal with, plus some information about them for use by our software configuration management toolkit. After having made about 200 attempts† to finally fix the set of attributes necessary to hold sufficient information about software objects, we gave up and invented the user defined attributes instead. One important inspiration for user defined attributes was the environment concept known from the UNIX shell. Once available, the environment is extended by some new variable almost each time a new tool is added to the system.

So we defined AFS as an internal interface. While writing our first AFS applications, we got aware of the power and the insufficiencies of the mechanism of application defined attributes. We believe that many applications might benefit from this concept, and that it is too valuable to be just an internal interface. Further development of AFS will primarily go into the direction described in the previous chapter. We are curious, where this way will lead us to.

6. 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.

† Actually we made only four or five, but 200 sounds better

References

1. 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.
2. Stuart I. Feldman, "MAKE - A Program for Maintaining Computer Programs," *Software - Practice and Experience*, vol. 9,3, pp. 255-265, March 1979.
3. 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.
4. 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.
5. Axel Mahler and Andreas Lampen, "A Toolkit for Software Configuration Management," *Proceedings of the Spring 1988 EUUG Conference*, pp. 185-202, European Unix systems User Group, London, UK, April 1988.
6. John R. Nestor, "Toward a Persistent Object Base," *Lecture Notes in Computer Science*, pp. 372-394, Springer Verlag, Trondheim, Norway, June 1986.
7. Wolfgang Obst, "Delta Technique and String-to-String Correction," *Lecture Notes in Computer Science*, pp. 69-73, Springer Verlag, Berlin, September 1987.
8. Marc J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 364-370, 1975.
9. Walter F. Tichy, "RCS - A System for Version Control," *Software - Practice and Experience*, vol. 15,7, pp. 637-654, July 1985.
10. 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.
11. Stanley B. Zdonik, "Version Management in an Object-Oriented Database," *International Workshop on Advanced Programming Environments*, pp. 405-422, Springer Verlag, Trondheim, Norway, June 1986.