On the design of the Amoeba Configuration Manager

Erik H. Baalbergen
Kees Verstoep
Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

The program *Amoeba Make*, or *Amake*, is being designed to fulfil the need of a *make*-like configuration manager capable of exploiting the potentials of the Amoeba distributed operating system. The major design goal is to create a software configuration manager that is both easy to use and efficient. The specification and maintenance of a large configuration should be easy, and should be automated as much as possible. Furthermore, the build process should exploit Amoeba's capabilities and resources when creating or updating a target. In this paper we show how a smart file server can contribute to *Amake*'s efficiency. We also show how a declarative configuration description allows *Amake* to take full advantage of parallelism and to determine the commands needed for building and maintaining targets.

1. INTRODUCTION

The program *Amake* was designed to fulfil the need of a *make*-like configuration manager that tries to overcome *make*'s inability to maintain large and complex systems in a convenient way, and that is capable of exploiting the potentials of the Amoeba distributed operating system.

Make[8] was introduced in 1979 as part of the UNIX™ programming environment. Its task is to maintain small and medium-sized programs. *Make* uses the user-supplied description file (called the *Makefile*), the file names and last-modified times of the files involved, and built-in rules, to maintain the system as described in the *Makefile*. To assure that the program is up-to-date with its sources, the user simply types ‘‘make’’. *Make* then checks all dependencies between generated (intermediate) files and the components that were used to generate them, and tries to update the final *target* with as few operations as possible. We assume the reader is familiar with *make* and *make*-related terminology.

Throughout *make*'s existence, people have used it for maintaining both small and large projects. Since programs have grown and the tools used for developing these programs have become fairly complex, *make* has shown its deficiencies for maintaining such systems. Most critique on the *make* program stems from the use of make for maintaining large and complex systems, for which *make* was not intended.[31] Feldman already said:[8] *Make* is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. Therefore we should not criticize the *make* program as such; it has proved its usefulness in the environment it was aimed at. But people have started to use the tool for maintaining large systems, and as such criticisms have arisen.

Due to *make*'s inability to maintain large systems and to handle parallelism, many new *make*-like programs have seen the light of day.[1, 7, 9-11, 19, 27, 32] These *make*-clones and *make* derivates solve a subset of the problems in ad hoc ways, and place the burden of writing complex description files on the programmer's shoulders. Another development is the integration of configuration management tasks into software development environments.[3, 6, 13, 14, 16, 17] Our purpose, however, is to design a configuration manager that runs stand-alone, and that is as independent from its environment as possible.

Instead of listing all *make*'s deficiencies, we limit ourselves to the ones that we wish to solve in *Amake*. The most important problem is that of writing and maintaining description files, which is boring and error prone. The programmer often has to twist and turn to express his needs. The difficulties that we try to overcome are:

- Implicit rules are based on file name suffixes only.
- *Make* does not combine implicit rules. The user has to be aware of and specify intermediate steps and targets.
- Multiple-output command blocks need a recursive invocation of *make.*
- Integrity is hard to maintain. A target can only be up-to-date if its dependency list is kept up-to-date. Furthermore, since macro definitions within a *Makefile* may change, *Makefile* itself should be in the dependency list of each target that is made using a macro definition. Worse yet, macro definitions can be specified as command-line options to *make.*
- *Makefile*s often contain repetitive and redundant information.
- *Make* assumes that a target's command block indeed produces the target. This may lead to obscure description file constructions.

A second important problem is that *make*'s notion of a target being ''out-of-date'' with respect to its dependents is based on the notion of time. In practice, time is a well-defined notion on single computers, but is hard to maintain in a distributed environment. Note that *make*'s time notion in combination with fast compilations requires a fine-grained time measurement. A third problem is that the original Feldman *make* does not exploit parallelism. True parallelism can be exploited in today's systems, and can be employed to significantly speed up the build process.[1]

Section 2 discusses the relevant aspects of Amoeba. In section 3 we present the purposes and the design of *Amake*. An example of the build process performed by *Amake* is discussed in section 4. Related work is discussed in section 5.

2. AMOEBA

Amoeba[20, 24] is an object-based distributed operating system originally designed at the Vrije Universiteit and now being further developed there and at the Centre for Mathematics and Computer Science (CWI), both in Amsterdam. The Amoeba architecture consists of four components. First are the *workstations*, one workstation per user, which run window management software, and on which the users can perform tasks that require fast interactive response, such as editing.[25, 30] Second are the *pool processors*, a groups of CPUs that can be dynamically allocated as needed, used, and then returned to the pool.

Third are the *specialized servers*, such as directory and file servers. Fourth are the wide-area network *gateways*, which are used to link local Amoeba systems at different sites into a single, uniform system.[23, 26]

Objects in Amoeba are instances of abstract data types such as files, directories and processes, and are managed by server processes. A client process carries out operations on an object by sending a request message to the server process that manages the object. While the client blocks, the server performs the requested operation on the object. When the operation finishes, the server sends a reply message back to the client, which is then unblocked. This request/reply message exchange is called a *transaction* in Amoeba.

The objects in Amoeba are named and protected by *capabilities*.[33] The capabilities, combined with transactions, provide a uniform interface to all objects in Amoeba. A capability is composed of four fields as shown below.

Server Port	Object Number	Rights	Check Field

The *server port* is a sparse address identifying the server process that manages the object. The *object number* is an internal identifier the server uses to distinguish among its objects. The *rights field* is a bit string telling which operations on the object are permitted by the holder of the capability. A capability is protected against forging and tampering by the *check field*, a large number. Since capabilities are clumsy to work with directly, they can be stored in *directories*, managed by the *directory service*. A directory is effectively a set of <ASCII string, capability> pairs. Common operations on directories are entering and deleting directory entries, and mapping ASCII names onto the corresponding capabilities.

3. AMOEBA MAKE

Given the deficiencies of *make* on the one hand, the parallelism available and the absence of a suitable ‘‘time’’ concept in Amoeba on the other hand, we felt the need to design a new configuration manager for Amoeba. The configuration manager is called *Amoeba Make*, or *Amake*. Our major design goal is to make *Amake* easy to use and to exploit Amoeba’s potentials. A less important goal is to allow *Amake* to be used on other, possibly distributed, systems.

In more detail, we have the following design goals. First, we adopt the basic philosophy of *make*, in which model a target is created and kept up-to-date according to the description file. Second, we want to exploit the parallelism and distribution offered by Amoeba. Experiments have shown that parallelism and distribution can lead to a significant reduction of the time needed for building a target.[1, 2] Finally, we want *Amake* to be rather ‘‘intelligent’’, in order to provide a tool that is easy to use. The ideal case would be to have a tool that, given a set of sources, builds and maintains a specified target with as little information from the user as possible.

Amake acts as a configuration manager; each time a programmer wants to ensure that targets are up-to-date according to a description file, he invokes *Amake*. *Amake* then runs the build process using information from the description file, the *Amakefile*. As soon as *Amake* finishes successfully, the configuration will be consistent with its description. From then on, the programmer can alter sources, and run *Amake* again to regain consistency.

Amoeba Make has to perform three tasks. Given a description of the source and target objects, it initially has to *determine* the list of commands needed for creating the targets. Next, *Amake* has to reduce the number of commands, since the results from commands from previous builds can be used and applied. Finally, it has to execute the remaining commands to update a target as efficiently as possible. We believe that the current design of *Amake* in combination with Amoeba serves these three needs and meets the design goals.

3.1. Command determination

To enable *Amake* to derive the commands needed for constructing a target, the programmer needs to specify at least the corresponding source files and the set of programs (or tools) that *Amake* may consider. The input for *Amake* is purely declarative, since the user specifies all components needed to build a target, without indicating *how* to build it. The declarative approach offers the advantage that the user is not concerned with details of when to run what program with which inputs. Furthermore, *Amake* can compose its own blue-print of the build process, and thus use any obscure system-specific features without the user noticing it. This blue-print is called the *configuration graph* of the system. The build process fills in the components of the graph and checks if the graph is up-to-date with respect to the current set of source objects.

Amake considers its sources, targets, and any intermediate objects to be attributed objects. The sources, targets, and (most) intermediate file objects are represented as typeless files, without a means to store out-of-band information. Typed file systems[22] are not widely used, and attributed file systems still are rare.[15] However, *Amake* attaches semantic knowledge to the objects using a set of, possibly valued, attributes, such as *TYPE=...* and *NAME=....* In an *Amakefile*, each object has an attribute *NAME=id*, and is referred to by *id*. The attributes are either declared explicitly or derived. The user can assign attributes to the file objects explicitly. *Amake* is able to derive or compute other attributes of an object according to *attribute derivation rules*, using the already known attributes and probably some tools. For example, an attribute derivation rule may express that any object with a name that ends with ‘‘.c’’ is a C-source file, and therefore has attribute *TYPE=C-SOURCE*.

Having attributed files instead of (strongly) typed files has the advantage that semantic knowledge can be attached to objects dynamically, and that the typing scheme is less severe. Semantic knowledge might include the fact whether an object is generated or not (*GENERATED=yes/no*), is suitable for being sent to the line printer (*PRINTABLE=yes/no*), or any other information that is relevant when applying operations to an object. In *Amake*, the attribute *CAPABILITY=cap* is attached to each object, and is used in the build process as we will explain further on.

The second source of information in *Amake* is the set of tools that can be applied in the build process. Tools are declared in *tool definitions* that express the characteristics and behavior of the tools. The requirements on the inputs and characteristics of the outputs are specified by means of attributes and attribute values that must be attached to the input objects or become attached to the output objects. A tool definition describes the commands to be executed, without specifying *when* to do this. It is up to *Amake* to decide when to invoke a particular tool and execute the corresponding commands. Furthermore, a tool invocation records its own *actual inputs*, and updates the configuration graph if needed.

It is possible that the actual inputs of a tool are not known when the tool has to be applied, since their names are computed from the contents of the *explicit inputs*. Explicit inputs are the inputs that are named explicitly in a tool invocation. Consider cc -c f.c, where the explicit input *f.c* may #include a set of files directly and indirectly. *Make* requires the included files to be specified as dependents of the target that is the result of the command. Tools like *makedep* automate the process of collecting the dependents, but are often clumsy to work with from within a *Makefile*. In *Amake*, a tool itself produces a list of files that were indeed read during the execution of the commands. The list of input files is merged into the outputs' dependency lists. Each dependency list is the list of actual inputs of the tool that produced the outputs. Note that this mechanism triggers the tool if any of the real input files, included either directly or indirectly by an input, has changed since the last *Amake* run. Whenever a new file is included by one of the current input files of a tool, the input file becomes changed, and the tool is triggered. The tool itself takes care of appending the new file to its input list, thus updating the configuration graph.

Tools can be invoked in several ways. The normal use, as in conventional systems, is to invoke a tool *imperatively*, to produce the desired outputs or effects immediately. An example is make print. Another strategy is *demand-driven* (*backward chaining*) tool invocation. The execution of the corresponding commands is triggered if one of its outputs is needed, without directly naming the tool. This is the way UNIX *make* works when requested to create or update a target. A third strategy is *data-driven* (*forward chaining*, *optimistic*) tool invocation. As soon as the inputs of a tool become available or change, the commands are triggered for execution. *Amake* will use the tool definitions in a forward chaining manner, but sometimes applies backward chaining to build subtargets. Tools are used imperatively in commands like Amake clean and Amake print, and within other tool definition command parts, for example to rename or move files.

Since most tool definitions are common, there exists a standard prelude that can be included in a user's *Amakefile*. The standard set contains tool definitions for compilers, code generators, linkers, and common file operations, such as copying, renaming, etc. The standard prelude contains the system-specific definitions, hidden inside tool definitions; system-dependent information can thus be kept out of the *Amakefile* itself. *Amakefiles* can therefore be kept rather system independent.

3.2. Command reduction

To speed up the build process, *make* tries to detect if a target is still up-to-date by looking at the file modification times of the target's dependents. If the target is newer than all its dependents, there is no need to regenerate the target. Since there is no suitable concept of time in Amoeba, we have adopted another mechanism to check targets against their dependents. To do this, we use the *CAPABILITY* attributes of the files involved. The value of the *CAPABILITY* attribute is the capability of the current contents, or *value*, of the file.

We assume the files involved in *Amake* to be stored on the *Bullet server*.[24] The Bullet server is a fast file server with an immutable-file store, with as principal operations *CREATE_FILE* and *READ_FILE*. The *CREATE_FILE* operation stores the given value on the file store, and returns a capability for further references, such as read requests. *READ_FILE* returns the value corresponding to the given capability. An advantage of the immutability of files is that processes always see an internally consistent file. When an application wants to change a file, it reads the complete file into its memory, makes the required changes, and stores the new file, thus getting a new capability. The old file is left unchanged. The binding from Bullet capabilities to the Bullet files' contents is immutable; a given capability returns the same value, irrespective of time.

The link between a Bullet file, identified by its capability, and the symbolic name, used by programmers and *Amake*, is done via the directory service. The directory service maps the symbolic name onto the Bullet-file capability, representing the current value of the file. The value of a symbolically named file therefore can change, since the directory service allows entries to be added and deleted, and thus the binding of a symbolic name to a capability to change. To detect if the value of a symbolically named file has changed since a previous point, we need to check if the binding from its name onto the Bullet-file capability has changed. This can be done by remembering the capability at the previous point. Note that this is not possible in UNIX since the most primitive file identifiers the programmer can use, namely inode numbers, are bound to mutable values.

Amake keeps track of the name-capability bindings when invoking tools. At each tool invocation, the bindings from the names of the inputs and outputs, and possibly the program itself, to Bullet-file capabilities are stored. Outputs are inconsistent with the inputs if one or more of the name bindings have changed. If the user edits a file, and the file gets a new value, the Bullet-file capability will be different, and *Amake* will detect and propagate the change. The name-capability bindings of a configuration, in which all targets are up-to-date, are kept by *Amake* in a so-called *Statefile*.

Further reduction of the number of commands to be executed can be achieved by using the *fire wall* concept: a change in the source objects propagates as far as the (intermediate) targets change. Consider, for example, the change of a comment in a C-source file. The source file is changed, and thus requires recompilation of the source file into an object file. *Make* simply assumes that the contents of the object file now have changed, and decides to take the next step, namely linking the object modules. However, the object file may have the same contents as the previous version, in which case there is no need to proceed the build process. To be able to *reuse* previous file contents, the tools might compare the contents of the output files against those of the previous output files, and replace output files only if the contents have changed.

Instead of letting the tools take care of file reuse, we have chosen to have the file server provide the concept. We added a third operation, *COMPARE_CREATE*, to the Bullet server. When creating a Bullet file, the Bullet server can be given an additional capability that is used as a hint. This hint is the Bullet-file capability of the current value of a symbolically named file. The Bullet server then checks if the hint represents a value that is equal to the new value to be stored. If it does, the server does not create a new Bullet file, and returns the hint as the capability of the Bullet file. In the C-source file example, the newly generated object code is stored on the Bullet server with the capability of the previously generated object-code file as hint. *Amake* checks afterwards if the binding of the object file name to its capability has changed. If it has not changed, no further action is required because the inputs of the linker have not changed.

3.3. Command execution

Tool definitions provide an abstraction of the programs and actions needed to accomplish a tool invocation. As such, tool definitions provide a system independent method of building a configuration graph. A tool definition contains a description of the actions that implement the tool. As soon as *Amake* has decided to invoke a tool, the corresponding actions need to be executed. We currently have two interfaces in mind. Since *Amake* may run on UNIX the Shell command language can be used to specify actions.[4] In Amoeba the *Bulletin Board* server can be used.[28] To execute a command, *Amake* fills in a *form* and invokes the Bulletin Board service, which takes care of the execution of the requested command on the form.

Parallelism can be exploited to further speed up the build process.[1] Like other parallel *make*s,[5, 9, 27, 29] *Amake* can execute the command blocks in parallel if there is no *depends-on* relation among them. Synchronization is done at the points where the results of the tools are needed. Mutual exclusive execution of tools is needed if the tools use fixed-named intermediate files. *Yacc*, for example, uses *y.tab.c*. Running multiple *yacc* invocations in the same environment is therefore dangerous. Mutual exclusion is taken care of by providing semaphores.

4. AMAKE AT WORK

In this section we present a simple example of *Amake*'s build process. We consider a small compiler with one *YACC*[12] source file containing the parser code, *parse.y*, one *LEX*[18] input file, *scan.l*, describing the lexical scanner, two C-source code files, *comp.c* and *defs.h*, and a library, *comp.a*, containing some utility routines, which may be used in the compiler. Furthermore, we assume that *scan.l*, *parse.y*, and *comp.c* include *defs.h* and *parse.h*, the latter of which is produced by running *yacc*. To enable *Amake* to produce the desired executable program, *comp*, out of the source files, we need to specify the tool definitions of the tools that can be applied during the build process. An *Amakefile* that describes our configuration is shown in Figure 1:

```
include std_amakefile .
target comp(TYPE=PROGRAM) :
        parse.y scan.l comp.c defs.h comp.a .
define CFLAGS = -DSUN .
```

Fig. 1. An example *Amakefile.* The tool definitions and attribute derivation rules are obtained from std_amakefile, which is included. The second line specifies that comp is an executable program (indicated by the attribute *TYPE=PROGRAM*) and can be created using the five given source objects and the definitions and rules obtained from std_amakefile. The last line specifies a configuration flag, which is used in C compilations. Configuration flags can also be specified per source file, thus overruling possible global definitions or enabling per-file compilation flags.

Using the *Amakefile* from Figure 1, *Amake* determines the configuration graph, as is shown in Figure 2. The resulting configuration graph is used in further builds.

The *Statefile* contains after a build the (name, capability) bindings of the input and output files, together with tool options, of each tool invocation. For example, the state information of the *cc -c* command to create *comp.o*, is shown in Figure 3. As soon as at least one of *comp.c*, *parse.h*, *defs.h*, or the definition of *CFLAGS* changes, *comp.o* must be remade. The command is also triggered if *comp.o* has changed without using *Amake*. Suppose that, for example, the programmer changes *parse.y*. Since the (*parse.y*, cap) circlerad = 0.5 C2: circle "scan.l" move 1i C1: circle "parse.y" move 1i C4: circle "defs.h" move 1i C3: circle "comp.c" move 1i C5: circle "comp.a" Byacc: box "yacc" at C1 + (0, -1.5i) Blex: box "lex" at C2 + (0, -1.5i) C7: circle "parse.c" at Byacc + (-0.7i, -1.5i) C8: circle "parse.h" at Byacc + (0.7i, -1.5i) C9: circle "scan.c" at Blex + (0, -1.5i) arrow from C1 to Byacc chop C1.rad chop Byacc.ht/2 arrow from C2 to Blex chop C2.rad chop Blex.ht/2 arrow from Byacc.s+(-0.2*Byacc.wid,0) to C7 chop 0 chop C7.rad arrow from Byacc.s+(0.2*Byacc.wid,0) to C8 chop 0 chop C8.rad arrow from Blex to C9 chop Blex.ht/2 chop C9.rad B3: box "cc -c" at C3 + (0, -4.5i) B4: box "cc -c" at C7 + (0, -1.5i) B5: box "cc -c" at C9 + (0, -1.5i) arrow from C3 to B3 chop C3.rad chop B3.ht/2 arrow from C7 to B4 chop C7.rad chop B4.ht/2 arrow from C9 to B5 chop C9.rad chop B5.ht/2 C10: circle "comp.o" at B3 + (0, -1.5i) C11: circle "parse.o" at B4 + (0, -1.5i) C12: circle "scan.o" at B5 + (0, -1.5i) B6: box "ld" at C4 + (0, -8i) C13: circle "comp" at B6 + (0, -1.5i) arrow from B3 to C10 chop B3.ht/2 chop C10.rad arrow from B4 to C11 chop B4.ht/2 chop C11.rad arrow from B5 to C12 chop B5.ht/2 chop C12.rad arrow from C10 to B6.nw + (0.6*B6.wid, 0) chop C10.rad chop 0 arrow from C11 to B6.nw + (0.4*B6.wid, 0) chop C11.rad chop 0 arrow from C12 to B6.nw + (0.2*B6.wid, 0) chop C12.rad chop 0 spline -> from C5 down 6i then to B6.nw + (0.8*B6.wid, 0) chop C5.rad chop 0 arrow from B6 to C13 chop B6.ht/2 chop C13.rad arrow from C8 to B5.n + (0.2*B5.wid,0) chop C8.rad chop 0 spline -> from C4 down 3.5i to B5.n + (0.4*B5.wid, 0) chop C4.rad chop 0 arrow from C8 to B4.n+(0.2*B4.wid,0) chop C8.rad chop 0 spline -> from C4 down 3.6i to B4.n+(0.4*B4.wid, 0) chop C4.rad chop 0 arrow from C8 to B3.n+(-0.4*B3.wid,0) chop C8.rad chop 0 arrow from C4 to B3.n+(-0.2*B3.wid,0) chop C4.rad chop 0 **Fig. 2**. The configuration graph constructed by *Amake* with source objects *scan.l*, *parse.y*, *defs.h*, *comp.c*, and *comp.a*. The target is *comp*. Circles represent file objects, and boxes represent tool invocations. The solid arrows display the structure of the initial configuration graph before any command is executed. A solid arrow from a file object to a tool invocation indicates that the file object is an explicit input of the tool. The dashed arrows represent the links that were added during the first build. The actual inputs of a tool are the file objects from which an arrow, either solid or dashed, is drawn to the tool invocation.

binding has changed, the *yacc* command is triggered. After *yacc* has finished, *parse.c* and *parse.h* may be bound to new capabilities. Now assume that *parse.c* indeed has changed, but *parse.h* has not. *Make* assumes that both files have changed, thus triggering rebuilds of *scan.o*, *parse.o* and *comp.o*. *Amake* notices that only *parse.c* has changed, thus triggering a rebuild of *parse.o*. However, *parse.h* is still bound to *cap2*. Therefore, there is no need to rebuild *comp.o* or *scan.o*. Note that any redefinition of *CFLAGS* also triggers the *cc -c* commands.

(comp.c, *cap1*)
(parse.h, *cap2*)
(defs.h, *cap3*)
CFLAGS=-DSUN
(comp.o, *cap4*)

Fig. 3. The state information concerning object *comp.o* directly, according to the *Amakefile* from Figure 1.

5. RELATED WORK

Many new stand-alone *make* programs and preprocessors have been developed during the last decade. These *make*s provide the programmer with extra facilities for describing configurations, but the description files have become fairly complex. They still require the programmer to specify intermediate targets; few of them provide facilities for building configuration graphs out of source dependencies, without intermediate targets being specified. The *Fourth Generation Make*[9] allows a target's direct dependents to be the source files that build up the target. Dependencies are then generated for C, *yacc*, and *lex* files according to hard-coded rules. *Mkmf*, part of the *SPMS* system,[21] creates a *Makefile* out of set of source files, a set of macro definitions, and a table containing mappings from filename suffixes to file types.

The *Pmak* configuration manager, developed as part of the *7001* project from *IST*,[34] provides a collection of script generators for generating *Pmakfiles* out of short descriptions. For example, the *pmcmd* command deals with C, *yacc*, and *lex* files. A significant improvement in *Amake* is that the knowledge about both tools and file types is not hardcoded. Tool definitions and attribute derivation rules allow users to tailor their own configuration management contexts.

The development of integrated software development environments leads to important improvements to the process of configuration management. Files are no longer typeless values stored in a file system, but have become entities in a data base or objects in an object base. Knowledge about file objects is kept with the objects themselves, and knowledge about tools, configurations, and relationships among the files, is expressed by rules, stored in a rule base. Version control, concurrent software development, and other software development activities, are integrated within a single programming environment.

In the *Cosmos* distributed programming environment,[22] a file is a typed object, which is an instance of a class. Rules use the type information for deriving objects and relationships among objects, for checking the proper usage of tools, and for maintaining consistency within a context. *Shape* is a *make* that uses the *Attributed File System*, *AFS*,[15, 19] for storing file objects. A set of attributes is attached to each file object maintained in *AFS*. The attributes are used in the selection of file objects when *Shape* is requested to compose a configuration.

Adele[3] uses a program data base to maintain a system. Files are attributed entities within the program data base. The attribute information is used for specifying configurations, according to which systems are composed and reconstructed. The *DSEE* Configuration Manager[16, 17] uses a system model, which is a set of dependencies among objects, and a *configuration thread*, which provides selection information, to compose a *version description*. The version description is then used in the build process to search the *derived-object pool*; if a desired object is not present, it is created and stored in the derived-object pool along with the version description used for creating it. Caching of previous results is achieved through the derived-object pool.

Marvel[13, 14] is an intelligent software development and maintenance assistant. It uses an active object base to manage its attributed file objects. A set of precondition-action-postcondition rules allows tools to be invoked, and activities to take place automatically when the object base is updated. The rules allow actions to be done in both forward and backward chaining fashion; the decision when to activate tools and actions is user-tailorable.

Amake is being developed as a stand-alone configuration manager, and as such does not resemble an integrated software development environment. *Amake* borrows and combines useful concepts from existing environments, like exploiting knowledge about file objects, automatic configuration building, result caching, and rule-based processing.

6. CONCLUSIONS

We expect the smart file server and the declarative way of describing things will indeed contribute to *Amake*'s performance and user-friendliness. The capabilities offered by Amoeba are exploited well enough to provide the user a *make*-like tool that can compete with other rather intelligent, and often integrated, configuration managers. *Amake* is currently being developed to run on Amoeba, but we expect to make it run on other systems as well; the least requirement is that *Amake* is able to uniquely (both in time and space) identify a file's contents. In UNIX, for example, we can use information from a file's inode (e.g., combination of inode number, device number, and file modification time) to identify the contents of a file. Unfortunately, UNIX offers possibilities to indeed change inode information explicitly. (E.g. via the system call *utime*.) In the UNIX implementation, we assume that the user would not tamper inodes.

7. ACKNOWLEDGEMENTS

We would like to thank Dick Grune, Henri Bal, Robbert van Renesse, and Frans Kaashoek for their critical reading of the manuscript and their valuable suggestions.

8. REFERENCES

References

1. E. H. Baalbergen, ''Design and Implementation of Parallel Make,'' *Computing Systems*, vol. 1, no. 2, pp. 135-158, Spring 1988.
2. E. H. Baalbergen, ''Parallel and Distributed Compilations in Loosely-Coupled Systems: A Case Study,'' *Proc. Workshop on Large Grain Parallelism*, Providence, RI, October 1986.
3. N. Belkhatir and J. Estublier, ''Experience With A Data Base Of Programs,'' *ACM SIGPLAN Notices*, vol. 22, no. 1, pp. 84-91, January 1987.
4. S.R. Bourne, ''UNIX Time-Sharing System: The UNIX Shell,'' *Bell System Technical Journal*, vol. 57, no. 6, pp. 1971-1990, 1978.
5. B. Cmelik, ''Concurrent Make: The Design and Implementation of a Distributed Program in Concurrent C,'' in *Concurrent C Project*, AT&T Bell Laboratories, Murray Hill, NJ, 1986.
6. J. Donahue, ''Integration Methods in Cedar,'' *SIGPLAN Notices*, vol. 20, no. 7, pp. 245-251, July 1985.
7. V.B. Erickson and J.F. Pellegrin, ''Build—A Software Construction Tool,'' *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 6, pp. 1049-1059, July-August 1984.
8. S.I. Feldman, ''Make—A Program for Maintaining Computer Programs,'' *Software—Practice and Experience*, vol. 9, no. 4, pp. 255-265, April 1979.
9. G.S. Fowler, ''The Fourth Generation Make,'' *Proc. USENIX Summer Conference*, pp. 159-174, Portland, Oregon, June 1985.
10. E. Hirgelt, ''Enhancing Make or Re-inventing a Rounder Wheel,'' *Proc. USENIX Summer Conference*, pp. 45-58, Toronto, Canada, June 1983.
11. A. Hume, ''Mk: A Successor to Make,'' Computing Science Technical Report No. 141, AT&T Bell Laboratories, Murray Hill, NJ, November 1987.
12. S.C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, NJ, July 1978.
13. G.E. Kaiser, N.S. Barghouti, P.H. Feiler, and R.W. Schwanke, ''Database Support for Knowledge-Based Engineering Environments,'' *IEEE Expert*, Summer 1988 .
14. G.E. Kaiser, P.H. Feiler, and S.S. Popovich, ''Intelligent Assistance for Software Development and Maintenance,'' *IEEE Software*, pp. 40-49, May 1988.

15. A. Lampen and A. Mahler, ‘‘An Object Base for Attributed Software Objects,’’ *Proc. EUUG Autumn Conference*, pp. 95-105, Cascais, October 1988.

16. D.B. Leblang and R.P. Chase, Jr., ‘‘Computer-Aided Software Engineering in a Distributed Workstation Environment,’’ *SIGPLAN Notices*, vol. 19, no. 5, pp. 104-112, May 1984.

17. D.B. Leblang and R.P. Chase, ‘‘Parallel Software Configuration Management in a Network Environment,’’ *IEEE Software*, pp. 28-35, November 1987.

18. M.E. Lesk and E. Schmidt, *Lex—A lexical Analyzer Generator*, Bell Laboratories, Murray Hill, NJ, 1978.

19. A. Mahler and A. Lampen, ‘‘A Toolkit for Software Configuration Management,’’ *Proc. EUUG Spring Conference*, pp. 185-202, London, April 1988.

20. S. J. Mullender and A. S. Tanenbaum, ‘‘The Design of a Capability-Based Distributed Operating System,’’ *The Computer Journal*, vol. 29, no. 4, pp. 289-300, March 1986.

21. P.J. Nicklin, ‘‘The SPMS Software Project Management System,’’ in *UNIX 4.2 BSD Programmers Manual, Vol. 2C, Part 3*, pp. 137-178, August 1983.

22. J.R. Nicol, ‘‘Operating System Design for Distributed Programming Environments,’’ *Ph.D. Thesis*, University of Lancaster, UK, October 1986.

23. R. van Renesse, J. M. van Staveren, J. Hall, M. Turnbull, A. A. Janssen, A. J. Jansen, S. J. Mullender, D. B. Holden, A. Bastable, T. Fallmyr, D. Johansen, K. S. Mullender, and W. Zimmer, ‘‘MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System,’’ *Proc. of the EUTECO 88 Conf.*, pp. 823-831, North-Holland, Vienna, Austria, April 1988.

24. R. van Renesse, J. M. van Staveren, and A. S. Tanenbaum, ‘‘The Performance of the Amoeba Distributed Operating System,’’ *Software—Practice and Experience*, vol. 19, no. 3, pp. 223-234, March 1989.

25. R. van Renesse, A. S. Tanenbaum, and G. J. Sharp, ‘‘The Workstation: Computing Resource or Just a Terminal?,’’ *Proc. of the Workshop on Workstation Operating Systems*, Cambridge, MA, November 1987.

26. R. van Renesse, A. S. Tanenbaum, J. M. van Staveren, and J. Hall, ‘‘Connecting RPC-Based Distributed Systems Using Wide-Area Networks,’’ *Proc. of the 7th Int. Conf. on Distr. Computing Systems*, pp. 28-34, West Berlin, September 1987.

27. E.S. Roberts and J.R. Ellis, ‘‘Parmake and Dp: Experience with a Distributed, Parallel Implementation of make,’’ *Proc. 2nd Workshop on Large-Grained Parallelism*, pp. 74-76, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1987. Available as Tech. Rep. CMU/SEI-87-SR-5

28. A. J. Schrander and J. W. van Otten, ‘‘The Bulletin Board Server: A Tool for Implementing Parallel Algorithms,’’ Master Thesis, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, February 1988.

29. Sequent, ‘‘DYNIX Make Manual Page,’’ in *DYNIX Programmer’s Manual—Revision 1.15*, August 1987.

30. G. J. Sharp, ‘‘The Design of a Window System for Amoeba,’’ Report IR-142, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December

1987.

31. P. Singleton, *Makefile Usage and Abusage: a Case Study*, Keele University, UK, 1986.

32. Z. Somogyi, ''Cake: a Fifth Generation Version of Make,'' *Australian Unix system User Group Newsletter*, vol. 7, no. 6, pp. 22-31, AUUGN, April 1987.

33. A. S. Tanenbaum, S. J. Mullender, and R. van Renesse, ''Using Sparse Capabilities in a Distributed Operating System,'' *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 558-563, Cambridge, MA, May 1986.

34. D.M. Tilbrook and P.R.H. Place, ''Tools for the Maintenance and Installation of a Large Software Distribution,'' *Proc. EUUG Summer Conference*, Florence, Italy, 1986.