Next Generation Software Configuration Management System

Nathan DeBardeleben
Stacey Dorsey
Kim Hazelwood
Jonathan Perry

June 19, 1998

Chapter 1

Abstract

As companies move towards larger software projects, management of these projects becomes a crucial issue. Software management can take on many meanings, from revision control to dependency tree specification. The dependency tree specification side of software management has been chiefly controlled by the UNIX *Make* command. This lightweight, text-based program allows users to specify dependencies between files and then to execute the *Make* command. *Make* evaluates which files have changed since the last compile and then recompiles only the files that depended upon the changed files. This concept allows for only parts of large software projects to be recompiled, reducing the amount of time and resources consumed.

There are many properties that are missing from the UNIX *Make* environment. *Make* is not user-friendly. The dependency graph has no visual representation, which can make managing large projects difficult. More importantly, *Make* does not allow for files to be distributed across the network. This is becoming a more important part of the software development world as companies have branches throughout the world where many teams are working together on a project.

In this paper a design is presented of a Next Generation Software Configuration Management System. The traditional features of *Make* are incorporated and expanded upon. Background information in the configuration management field is presented to show some of the many uses of configuration management today and the future. Examples of *Make* are given so as to be referenced throughout and used to explain how the system presented here meets the requirements of being upwardly compatible with *Make*. Formal specification is employed to describe the design of the system and show that it allows for such functions as saving, loading, editing, viewing, and compiling a dependency tree. Specifications for converting a datafile from *Make* are also given to allow for a way for users to convert existing projects into the native format of this product. Some screenshots from a sample graphical user interface are also presented. These are to be used as guidelines for what the functionality and look of the implemented product should be.

There are many properties which were set to be accomplished in this project. Generality, self-realization, efficiency, portability, and software reuse were just a few of these. These terms are explained and then it is described how they were met with this design. Some sample code for communicating across the network is presented with test cases to facilitate the implementation. Finally, a glossary of new and important terms is presented as well.

While the *Make* program has ruled the UNIX environment for many years, it has many deficiencies which are becoming more and more apparent as programmers begin to work on larger projects. These must be corrected, and a design for the solution is presented herein.

Contents

List of Figures

Chapter 2

Background

"For programmers drowning in a sea of constantly changing source code, software configuration management promises to be a much-needed lifeboat." [Cronk, p. 45] As the software development process grows larger and more complex configuration management is almost a necessity to developing high quality products.

In a paper by Salvatore Salamone several reasons have been established of why having a Configuration Management system is a good idea. The reasons he lists are that it masks complexity of the network from users, reduces user options making LAN support easier, reduces training costs, facilitates changes when applications move or devices are added to the network, keeps users from trying to run programs their PCs cannot support, enforces uniform corporate image in customer service settings, and tightens securities. He goes on to give some good examples of situations where each of these apply [Salamone, p. 160].

With today's fast paced technology the process of software development is becoming a larger and a more complex process. To manage this complexity, Configuration Management systems have moved to the forefront for controlling and managing the process of software development. There are many desirable features of a Configuration Management system. Many of the features do exist in current CM systems although some exist more than others and no single system seems to contain all the features desired [Dart, pp. 3-4].

Three qualities currently exist in many configuration management systems. They are version control, a check-out/check-in facility, and a buffered-compare program [Buckley, p. 56]. "Version control is the ability to store multiple versions of the same file under controlled, restricted-access conditions. [Buckley, p. 56]" The check-out/check-in capability keeps more than one user from modifying the same file. The buffered-compare program compares the old and new files and provides as output "a complete delineation of the additions and deletions. [Buckley, p. 59]"

There are also four desirable capabilities which would be good to have in a CM system. These are establishing a standards-checking program, implementing an automated problem-reporting system, automating the generation of configuration status accounting reports and providing an automated metrics acquisition and reporting capability [Buckley, p. 59]. The features come from both the management and product side of the software development process. A standards-checking program checks all files to make sure they all conform to project standards and will generate a problem report if one is found that doesn't conform. The automated problem-reporting system is somewhat self explanatory. It will keep the project on track by getting problems reported quickly. The next desirable feature, automating the generation of configuration status accounting reports, gives details such as the status of change proposals and the revision levels of configuration documentation. This feature is desirable for larger projects such as in industry where the projects can become quite large. Finally, the desirable feature of providing an automated metrics acquisition and reporting capability provides reports on two types of metrics, those "relating to the software configuration management process itself, and project metrics providing insight into the software development process." [Buckley, p. 61]

The majority of these features do exist in different CM systems, for instance Aide-De-Camp (ADC) is an existing system which provides change sets for distribution of change. This system also integrates problem reports and change requests providing some of the features from the management side of the process.

Another existing CM system is Adele. Adele has basic features of data-modeling, interface checking, and representing families of products [Dart, p. 32]. "Since the system knows of the dependency graph, it can assist in composing a configuration. [Dart, p. 32]" Through this, the system can detect incomplete or inconsistent descriptions.

Another existing system offering some desirable features is CCC, Softool's Change and Configuration Tool. Also offering some management features, CCC provides conventions that go along with the waterfall model which is widely used in industry today. It offers online support of documentation standards and change requests [Dart, p. 32].

A system which offers more features is DSEE, Domain Software Engineering Environment. "DSEE provides derived object code management as well as source version control, system modeling, configuration threads, version selection based attributes, releases of configurations, system building, (reusable) object pools, task lists for tracking tasks to be done and those completed, and alerts for notifying users for certain events. [Dart, p. 34]" This system offers features from both the management side and the product side of the process.

Finally, just to mention a couple more systems of the many existing, we have RCS and DMS which are both version control systems, DMS being for files distributed across different platforms [Dart, pp. 11-12].

When considering the design of a Configuration Management system, one needs to think carefully of the criteria for choosing a software configuration management system. The system needs to offer features that are desirable in the market and will help a company develop the best software with the lowest overhead. It has been established that there are two main cost factors that must be considered when a company is deciding on a Configuration Management system, one "the hardware resources necessary to achieve acceptable performance" and two "the human resources needed to administer and maintain the SCM system. [Midha, p. 163]" These things must be considered to create a product that will excel in the marketplace and to have a product be chosen over other Configuration Management systems.

Looking toward the future of Configuration Management it has been discovered that the software development process upon which Configuration Management has been built is evolving to a new level. In industry today and in the recent past the Waterfall Model is widely used in the software development process but "the software industry however, is now tackling problems that the waterfall model cannot handle. [Bersoff, p. 106]" With these problems new models will be developed which will cause a change in needs from a Configuration Management system. New needs will arise and new systems will need to be developed to accommodate those needs.

The product described below is one which offers some of the established features of a CM system along with the advantage of product development with a team spread out across a network. The product's features are more towards the product side of the process and more away from the upper management side.

Chapter 3

Example Makefiles

One of the major objectives of the Configuration Management System is that it should be a logical extension of *Make*. Before moving any further in the specification of the system, it would be appropriate to introduce some sample makefiles that will be used as a reference point for further discussion. In addition, the scrutinization of these existing makefiles should ensure that the final design is complete and contains all of the functionality of the existing *Make* system. This approach seems to be a logical starting point for specifying the total system due to the fact that many design decisions must take into account the current functionality of makefiles while also providing network accessibility for the system.

The introduction of these makefiles will also provide a means for introducing and defining several key terms that will be frequently used from this point forward.

3.1 A Simple Makefile

The makefile shown next in Figure 3.1 was taken from the GnuMake Online Manual. This makefile introduces the dependencies surrounding a sample executable, called EDIT. Although this example is clearly referring to a system programmed in the C programming language, the final Configuration Management system should be language independent. The second example will portray an arbitrary system that is not based on any specific programming language.

As shown in Figure 3.2, EDIT has eight dependencies, each of which has a list of their own subsequent dependencies.

EDIT has associated with it a command for compiling and creating itself, as do EDIT's nine dependencies. Any node in the tree which has any commands associated with it is referred to as a goal. All other nodes are referred to as files. This concept will be explained in more detail in Section 3.3.

As the makefile from Figure 3.1 is scrutinized further, it becomes apparent that makefiles have uses other than strictly stating compilation rules. The *clean*

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
       cc -o edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Figure 3.1: A Simple Makefile

command contained within the makefile is independent of the EDIT executable and all of its sub-dependencies. Thus it can be said that the *clean* command is an independent goal. In this case, *clean* serves the purpose of removing all of the object files and therefore has nothing to do with compilation. This concept becomes an important factor in design decisions. The final implementation clearly must have the ability to deal with several, sometimes independent, goals.

The entire system contained in the makefile in Figure 3.1 can be graphically represented with the following model. Certainly, this model should be used as an example of the graphical representation that should be produced by the Configuration Management system and available to the user. The specific methodology for producing this model from the given dependencies is left to the programmer, as this methodology would vary depending on the language by which the representation is created. Figure 3.2 simply serves as a prototype of what may be implemented.

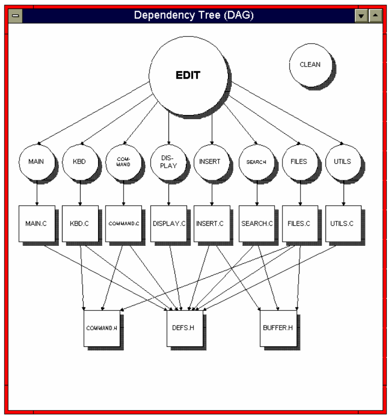


Figure 3.2: Directed Acyclic Graph : A Simple Makefile

Looking at this representation, it should be noted that each circle represents a goal and each square represents a file. In addition, the *clean* function is properly represented as an independent goal, with no connections to EDIT itself.

Although a traversal down the y-axis of this picture implies a deeper dependency and therefore a difference in the compilation order is implied, the same is not true for the x-axis. Nodes on the same level of the y-axis (i.e. main, kbd, command, etc.) can be compiled in any possible order, or all in parallel, if possible. Therefore, a traversal to the left or right on the x-axis carries no implications for compilation order.

Finally, although this example does not depict the case where a file has a goal as its dependant, that case is certainly allowed and will be clearly represented in the next, more complex, example makefile.

3.2 A More Complex Makefile

The purpose of the previous makefile was to provide a simple example, which can be referenced throughout the remainder of the text. The example lacked complexity, however, and it is therefore the goal of the next example makefile presented in Figure 3.3 to illustrate the extent to which the final Configuration Management system may be used. It further depicts the system's capability for dealing with such cases.

The example in Figure 3.3 contains three top-level goals : SYSTEM, PRINT, and SIZE. The *.fil extension is merely a generic representation of any given source file. Similarly, the *.obj extension refers to any object file. These extensions were contrived to illustrate the fact that this system is intended to be platform independent, therefore any arbitrary language may be implemented in the makefile.

While PRINT and SIZE are independent goals, the SYSTEM goal contains an intricate web of dependencies, which is best portrayed in the following graphical representation, Figure 3.4.

```
system : proj1 proj2 proj3
        cc proj1.obj proj2.obj proj3.obj

proj1 : alpha proj1.fil
        cc alpha.obj proj1.fil
proj2 : alpha proj2.fil
        cc alpha.obj proj2.fil
proj3 : stats.fil data.fil beta
        cc stats.fil data.fil beta.obj
alpha: alpha.fil help.fil
        cc -c alpha.fil help.fil
beta: beta.fil var.fil
        cc -c beta.fil var.fil
help.fil : link
        cc -o link.obj
beta.fil : link
        cc -o link.obj
link : psi.fil gamma.fil delta.fil
        cc -c psi.fil gamma.fil delta.fil
psi.fil : sigma.fil
        cc -c sigma.fil
gamma.fil : sigma.fil
        cc -c sigma.fil
delta.fil : sigma.fil
        cc -c sigma.fil
print:
        lpr -Plpp system.bmp
size:
        ls  | wc > dataset.size
```

Figure 3.3: A More Complex Makefile

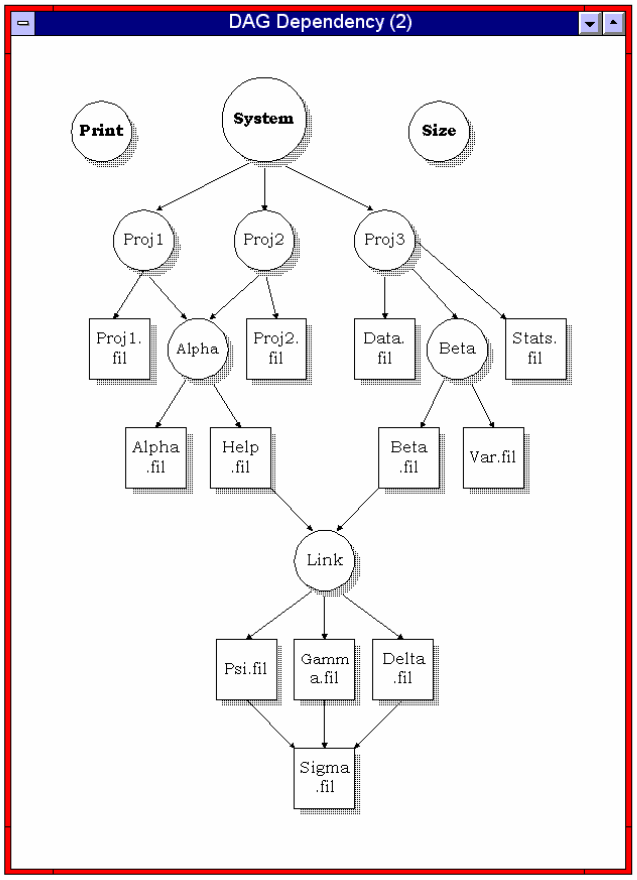


Figure 3.4: Directed Acyclic Graph : A More Complex Makefile

This complex set of dependencies will become useful when the compilation methodology is explained later in Section 4.7. For now, it should simply be noted that files have as dependants both files and goals, sometimes simultaneously; and this is also the case for goals.

3.3 Introduction of Terms

Throughout this document certain terms are used which are in need of definition. Figure 3.4 shows a graphical representation of the more complex makefile from Figure 3.3. In this drawing it can be seen that some nodes have multiple parents, such as *Link*, *Sigma.fil*, and *Alpha*. These nodes create a problem when viewing the dependency graph as a tree. It also makes a depth first search impossible because nodes would be hit multiple times. With these nodes spread out across the network, this could be an intensive and unacceptable task.

The graph is therefore what is termed *Directed Acyclic Graph*, or a DAG. There are formulas for converting a DAG into a spanning tree so that a depth-first search can be performed. In this way, no node would be hit more than once. However, other methods than this were used to solve this problem. These will be discussed more when the compilation algorithm is explained in Section 4.7.

All the objects in Figure 3.2 and Figure 3.4 are called nodes. These are objects which have parents and children and some form of identification. This identification can be through a name, a number, or another form but it is imperative that there be a way to distinguish a particular node from another. As can be seen from Figure 3.2 and Figure 3.4, there are two different shapes on the graphs. The round nodes are called *goals* and the square ones are termed *files.*

Goals are not physical entities. They are not located anywhere on the network. They cannot be copied, they have no date when they were last updated. However, they do have commands associated with them. These commands are executed when it is determined that a goal must be updated. This process will be explained in more detail in the specification of the Manager and the compilation procedure, in the Section 4.4.6.

Files, on the other hand, do have location. They can be copied, and most importantly, they have a date associated with them when they were last modified. This date is retrieved through the standard POSIX interface and allows for the system to determine if a file has changed since the last time it recorded its date. In this way, an extremely large system which takes a long time to compile for the first time would take considerably less then next time if only one file has changed. The system would recognize that only one file has changed and then update only the files which depended upon it. This concept is not a new one. It is used as the basis for the UNIX *Make* program. However, *Make* does not allow for files to be located throughout the network. This is overcome in the solution provided in this document as well as expanding on some of the other capabilities of *Make.*

Chapter 4

Comprehensive Description

4.1 File Locality

The main aspect of the configuration management system is that it allows for compilation of large software projects, such as the UNIX *Make* program. *Make* lacks the ability to have files stored on any network in the world. This property is becoming more and more important in the programming world today and managing these projects is becoming increasingly difficult with a tool such as *Make*.

In order to accomplish this goal of allowing software components to exist anywhere on the Internet, the Software Configuration Management system incorporates several remote/network features. The features will be extremely useful in the case where a large project exists whose files are dispersed across several networks, as is often the case in industry.

4.1.1 Non-Locality Resolution

All out-of-date files are brought to the local host for recompilation. Files are brought locally and compiled in order to avoid architectural incompatibilities. The CM system will include the ability to query the datestamp of a remote file in order to determine its status (up-to-date / out-of-date.) The result of this query will determine whether or not the remote source file should be brought to the local host for compilation. It should be noted that only remote *source files* are brought to the local host. Object files are not brought to the local host because the architectural differences between the remote server and the local host will typically render an object file useless.

As the datestamp on a file is queried, see Section 4.6, it is determined whether or not the object file stored in the local object directory is up-to-date. If it is determined that a remote file has been updated, that file will be brought to the local host and compiled. Following compilation, the local copy of the remote file will be destroyed in order to avoid naming conflicts or confusion in the datestamp system (see Section 4.6). Furthermore, the object file resulting from compilation will be stored in a preselected object file directory on the local host and will replace any outdated version of itself. This location will be determined by querying the OBJECT_FILE attribute located in the datafile, as described in Section 4.5.

4.1.2 File Retrieval

A goal of this product is to allow for any file to be located anywhere in the world as long as it is reachable. This is accomplished by using Uniform Resource Locators, or URLs. These are a standard way of locating a file and have some of the following protocols: Hyper Text Transfer Protocol (HTTP), File Transfer Protocol (FTP), and FILE. HTTP is commonly used for the World Wide Web, while FTP has been around for a long time and provides the ability issue some basic commands to transfer files to and from a specific site. FILE is another URL which points to a file on the local machine.

Some examples of the syntax of these three protocols follow.

```
http://www.clemson.edu/clemweb/index.html
ftp://shredder.parl.eng.clemson.edu/pub/dev/Main.java
file:/usr/X11RC/include/X11/Composite.h
```

The compilation operation that the user can invoke will traverse the dependency tree that they have provided. When it gets to a file, there must be a decision made if that file is a new file and therefore must be updated and recompiled (if it's a recompilable file). For this to be possible, the system must know the location of these files on the network using one of the protocols through a URL. The system can then locate that file and compare the date it was last modified with the date stored in the system that it was last modified. Since there is a record of all the dates last modified of all the files, in this way it can be determined if a file has changed.

Should it be determined that a file has been changed, that file is retrieved to the local system where it is used in the compilation process. After this process, the new date of the file is recorded in the system which now recognizes the file as being up to date.

4.2 Specification of the Network

The above discussion about how files are retrieved and dates checked requires specification. This is done through the specification of a simple network that is either connected, or not connected to a URL. What follows is the specification of that network.

Before the network can be specified, some sets must be established and defined for use.

There is a set of Uniform Resource Locators (URLs) that contain all the valid URLs that can be formed. This set contains many URLs that are not actually linked to any location, it is merely a set of all valid formatted URLs.

The set is named $[URL]$. There is another set of URLs that actually link to specific files. This set is named $[VALID\_URL]$. Note that:

$$VALID\_URL \in URL$$

The network can be specified by a simple state schema which determines if a connection is currently established.

$$\begin{array}{|l}
\multicolumn{1}{l}{Network} \\ \hline
connected : boolean \\ \hline
\end{array}$$

The initial state of the network must be established to be disconnected.

$$\begin{array}{|l}
\multicolumn{1}{l}{Network\_Init} \\ \hline
Network' \\ \hline
connected = false \\ \hline
\end{array}$$

For a connection to be made to a specific URL that URL must be specified to the connect schema. This operation will succeed if the URL is a valid one, however, it will fail if it is not.

A response type is now defined to allow for callers to understand the success or failure of an operation.

$$Response ::= SUCCESS \mid FAILURE$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Connect\_OK} \\ \hline
\Delta Network \\
someURL? : URL \\
res! : Response \\ \hline
connected = false \\
connected' = true \\
someURL? \in VALID\_URL \\
res! = SUCCESS \\ \hline
\end{array}$$

In the instance that the given URL to connect to is not valid, the operation must fail. This error condition is outlined next.

$$\begin{array}{|l}
\multicolumn{1}{l}{Connect\_ERROR} \\ \hline
\Xi Network \\
someURL? : URL \\
res! : Response \\ \hline
connected = false \\
someURL? \notin VALID\_URL \\
res! = FAILURE \\ \hline
\end{array}$$

$Connect \mathrel{\widehat=} Connect\_OK \land Connect\_ERROR$

Once connected, the date of the URL can be obtained. This operation will fail if the connection has not been established. There is a set of all dates $[DATE]$ of which the return type will be a member of.

$$\begin{array}{|l}
\multicolumn{1}{l}{GetDate\_OK} \\
\Delta Network \\
date! : DATE \\
r! : Response \\
\hline
connected = true \\
r! = SUCCESS \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{GetDate\_ERROR} \\
\Xi Network \\
r! : Response \\
\hline
connected = false \\
r! = FAILURE \\
\hline
\end{array}$$

$GetDate \mathrel{\widehat=} GetDate\_OK \land GetDate\_ERROR$

Should the date be satisfactory to warrant retrieval of the file, the file can be taken from the URL. This operation also requires being connected to the URL for success. The file that can be returned is from the set of all files $[FILE]$.

$$\begin{array}{|l}
\multicolumn{1}{l}{GetFile\_OK} \\
\Delta Network \\
file! : FILE \\
r! : Response \\
\hline
connected = true \\
r! = SUCCESS \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{GetFile\_ERROR} \\
\Xi Network \\
r! : Response \\
\hline
connected = false \\
r! = FAILURE \\
\hline
\end{array}$$

$GetFile \mathrel{\widehat=} GetFile\_OK \land GetFile\_ERROR$

Finally, there must be a way to disconnect from the URL. This operation does not necessarily require already being connected as it will perform nothing if there is no connection. Therefore the constraint will not be placed on the system.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Disconnect}} \\ \hline
\Delta \mathit{Network} \\ \hline
\mathit{connected}' = \mathit{false} \\ \hline
\end{array}$$

4.3 Revision Control

Commonly, a Configuration Management System will provide for revision control. A common example of this is RCS (Revision Control System). Revision Control Systems allow for a user to access older versions of a file, which have since been replaced by newer versions. RCS also ensures that no two users are able to edit a file at the same instant. This functionality, while often useful, is outside the scope of this project. It would be nearly impossible to verify that no two users are accessing a remote file at any given instant without a permanent connection to the remote system. This approach is not feasible given the fact that numerous remote systems may be accessed within one file structure, as in the case of the EDIT example. Theoretically, a permanent connection would need to be made to each of the remote systems. The connections would be required to monitor each of the files located in the remote system to detect when a user is attempting to revise a given file. This is clearly not feasible, and is therefore omitted from this Configuration Management system description. An appropriate alternative would be for each remote system to run their own version of revision control software.

4.4 Manager

The system is represented by what is termed the *Manager*. This manager could be thought of as a single process having memory where it can store and retrieve data. All input and output goes through this process and is handled strictly by this process. It controls the main operations that a user would want to perform, such as creating the dependency tree, changing the dependency tree in any way, compiling the project, and saving and loading the project.

4.4.1 Definitions

It is important for some definitions to be established so that the specification of the Manager will be clear. In the section describing sample makefiles, two graphs were shown, Figure 3.2 and Figure 3.4. Every object on the graphs are what are termed as *nodes*. Nodes are objects which have children and parents but nothing else. A *goal* is an extension of a node. It extends the structure of a node to allow for commands to be associated with a node. Goals do not exist anywhere. They are not representations of a file that exists on the network, and therefore do not have a location, or a last modified time. On the other hand, *files* are physical objects. Files are also a special type of node. However, a file does *not* have commands associated with it, it truly is a subclass of a node. Because files exist on the network, they have locations and times when they were last modified.

This structure is very important and employed in the compilation procedure. Because files do not have commands associated with them, if it is determined that a file must be updated, the file is copied to the local machine to be used when compiled by the parent goal. The resulting commands that are executed are done from the parent goal. This is so that a goal can depend upon many files and only upon successfully determining which ones have been updated does it execute its commands. This will be explained further in the compilation section, Section 4.7.

4.4.2 Manager State Schema

Before the state of the manager can be defined, it is important to establish all the sets and types which will be used throughout the specification.

There is a set of all possible nodes named $[NODE]$. There is also a set of all commands named $[COMMAND]$. The same set defined in the Network specification of URL is also defined here. Again, these URLs are merely URLs that are correctly formed and do not necessarily correctly link to a file. This set is named $[URL]$. There are also sets of all login names, $[LOGIN]$ and passwords $[PASSWORD]$ to be used when logging onto an FTP site, should it be required.

When the compile operation is to be performed, it is necessary for each node to have a state value. This state can take on exactly one of four values.

$$STATE ::= BUSY \mid NOT\_HIT \mid CHANGED \mid NOT\_CHANGED$$

The $BUSY$ state acts as a semaphore notifying that another node has already activated this node and therefore it cannot be reactivated. A node is activated when its parent node reaches it and asks it to determine if it has changed or not. The NOT_HIT state means that this node has not yet been activated and it can be activated now. Finally, the $CHANGED$ and $NOT\_CHANGED$ states express that the node has already been activated and no longer needs to be activated. In this case, it could have either changed, or not have changed. The operations which use the state of a node will be explained further in the Compilation Specification, Section 4.4.6.

Now the state schema for the manager can be defined. It has many data members which hold all the information about the design loaded, or more specifically, the dependency trees.

$$
\begin{array}{|l}
\hline
\textit{Manager} \\
nodes : \mathbb{P}\, NODE \\
goals : \mathbb{P}\, NODE \\
files : \mathbb{P}\, NODE \\
commands : \mathbb{P}(NODE ⇸ \mathbb{P}\, COMMAND) \\
locations : \mathbb{P}(NODE ⇸ URL) \\
datestamps : \mathbb{P}(NODE ⇸ Date) \\
nodeFiles : \mathbb{P}(NODE ⇸ \mathbb{P}\, NODE) \\
nodeGoals : \mathbb{P}(NODE ⇸ \mathbb{P}\, NODE) \\
logins : \mathbb{P}(URL ⇸ (LOGIN, PASSWORD)) \\
states : \mathbb{P}(NODE ⇸ STATE) \\
\hline
nodes = goals \cup files \\
\forall\, x \in goals \Rightarrow x \notin files \\
\forall\, y \in files \Rightarrow y \notin goals \\
a \mapsto b \in logins \Rightarrow somenode \mapsto a \in locations \\
c \mapsto d \in commands \Rightarrow c \in goals \\
e \mapsto f \in locations \Rightarrow e \in files \\
g \mapsto h \in datestamps \Rightarrow g \in files \\
i \mapsto j \in states \Rightarrow i \in nodes \\
\hline
\end{array}
$$

All the nodes in the system are made up of the goals and files together. No member can be of both goals and of files. If a specific URL is bound to a login name and password, then that URL must already be bound to a file location. In this way it is ensured that only URLs that have a *purpose* in the system have a login and password associated with them. Here it is defined that only goals can have commands, and that only files have URL locations and datestamps. Finally, it is shown that only valid nodes initialized in the system have states associated with them.

The initial state of the manager can now be established for consistency.

$$
\begin{array}{|l}
\hline
\textit{Manager\_Init} \\
Manager' \\
\hline
nodes' = \varnothing \\
goals' = \varnothing \\
files' = \varnothing \\
commands' = \varnothing \\
locations' = \varnothing \\
datestamps' = \varnothing \\
nodeFiles' = \varnothing \\
nodeGoals' = \varnothing \\
logins' = \varnothing \\
states' = \varnothing \\
\hline
\end{array}
$$

The initial state of the Manager must be shown not to violate the data invariants. The following proof accomplishes this simple task through set equality.

(1)	$nodes' = \varnothing$	(premise)
(2)	$goals' = \varnothing$	(premise)
(3)	$files' = \varnothing$	(premise)
(4)	$commands' = \varnothing$	(premise)
(5)	$locations' = \varnothing$	(premise)
(6)	$datestamps' = \varnothing$	(premise)
(7)	$nodeFiles' = \varnothing$	(premise)
(8)	$nodeGoals' = \varnothing$	(premise)
(9)	$logins' = \varnothing$	(premise)
(10)	$states' = \varnothing$	(premise)
(11)	$\varnothing = \varnothing \cup \varnothing$	((1), (2), (3), set ident.)
(12)	$nodes = goals \cup files$	(subst.)
(13)	$\forall x \in \varnothing \Rightarrow x \notin \varnothing$	((2), (3), set ident.)
(14)	$\forall x \in goals \Rightarrow x \notin files$	(subst.)
(15)	$\forall y \in \varnothing \Rightarrow y \notin \varnothing$	((3), (2), set ident.)
(16)	$\forall y \in files \Rightarrow y \notin goals$	(subst.)
(17)	$a \mapsto b \in \varnothing \Rightarrow somenode \mapsto a \in \varnothing$	((9), (5), set ident.)
(18)	$a \mapsto b \in logins \Rightarrow somenode \mapsto a \in locations$	(subst.)
(19)	$c \mapsto d \in \varnothing \Rightarrow somenode \mapsto c \in \varnothing$	((4), (2), set ident.)
(20)	$c \mapsto d \in commands \Rightarrow somenode \mapsto c \in goals$	(subst.)
(21)	$e \mapsto f \in \varnothing \Rightarrow somenode \mapsto e \in \varnothing$	((5), (3), set ident.)
(22)	$e \mapsto f \in locations \Rightarrow somenode \mapsto e \in files$	(subst.)
(23)	$g \mapsto h \in \varnothing \Rightarrow somenode \mapsto g \in \varnothing$	((6), (3), set ident.)
(24)	$g \mapsto h \in datestamps \Rightarrow somenode \mapsto g \in files$	(subst.)
(25)	$i \mapsto j \in \varnothing \Rightarrow somenode \mapsto i \in \varnothing$	((1)0, (1), set ident.)
(26)	$i \mapsto j \in states \Rightarrow somenode \mapsto i \in nodes$	(subst.)

Q.E.D.

It is important for every operation to have a return value of either success of failure. In this way, errors can be passed back to the calling procedures upon implementation. This will allow for the caller of these operations to handle errors accordingly. For instance, if a user tries to establish that a particular node A is a child of another node B when A is already a child of B, then an error will be passed back. This will allow an error message to be displayed, should the implementation call for that.

A response type is now defined to allow for callers to understand the success or failure of an operation.

$$RESPONSE ::= SUCCESS \mid FAILURE$$

4.4.3 System Schemas

There are several operations which are performed upon the system. Unlike the operations elsewhere, these operations do not affect nodes already established. *AddTopLevelGoal* is a schema which allows for the creation of root goals of the project. There may be many root goals, or in a simple case there might only be a single root. *GetGoals* and *GetFiles* are schemas which provide for a way to get the sets of all the goals and files in the system. This is commonly useful when constructing a graph where perhaps it is important to know how many nodes there will be in the total system. *ChangeNode* is an operation which allows for a system-wide change of a node into another node. While this operation does not allow for a user to convert a node into an already existing node, it will be useful in some instances. Finally, *DeleteNode* is very similar to *ChangeNode* is that it causes a system-wide change. In this case, however, a given node is deleted throughout the system, destroying all references to it everywhere.

Each operation will be more specifically described and defined now.

Users must be able to add what are termed *Top Level Goals* to the system. These goals are ones that have no parents, they are roots. It is important that this system have ways to add multiple top level goals, or have multiple roots. This is so that simple operations of commands that have no children can also be part of the system. It also allows for multiple trees and creates more than just an environment for compiling and managing a project but also a robust managing environment for several projects at a time. This could become an issue when several people are working on the same project at the same time, but need seperate dependency trees. For instance, one person could be managing a product A which has a dependency tree. Two other people could be managing products B and C, which also have dependency trees and perhaps depend on each others products. There could be three data files with three different trees, or simply one data file with three different trees. Of course this system would allow for both methods, but more importantly is the latter.

The operation that follows allows for adding of a top level goal. Of course the goal to be added cannot already be a node in the system. Upon successfully adding this node to the system, its children goals and files are set to be the emptyset.

$$\begin{array}{|l}
\hline
\mathit{AddTopLevelGoal\_OK} \\
\Delta \mathit{Manager} \\
\mathit{aNode}? : \mathit{NODE} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aNode}? \notin \mathit{nodes} \\
\mathit{nodes} = \mathit{nodes} \cup \mathit{aNode}? \\
\mathit{goals} = \mathit{goals} \cup \mathit{aNode}? \\
\mathit{aNode}? \mapsto \varnothing \in \mathit{nodeGoals}' \\
\mathit{aNode}? \mapsto \varnothing \in \mathit{nodeFiles}' \\
\mathit{res}! = \mathit{SUCCESS} \\
\hline
\end{array}$$

The operation to add a top level goal could fail should the node trying to be added already exist in the system. It is important to keep only one copy of all goals and one copy of all files in the system. It is also imperative that there are not nodes which are goals and files. The schema which describes this operations follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{AddTopLevelGoal\_ERROR}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{Manager} \\
\mathit{aNode}? : \mathit{NODE} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aNode}? \in \mathit{nodes} \\
\mathit{res}! = \mathit{FAILURE} \\
\hline
\end{array}$$

For completeness, the prepositional anding of these operations creates the complete operation to add a top level goal to the system. This convention is used throughout this specification and is explained here for clarity.

$$\mathit{AddTopLevelGoal} \mathrel{\widehat=} \mathit{AddTopLevelGoal\_OK} \land \mathit{AddTopLevelGoal\_ERROR}$$

The project contains many nodes. Some of those nodes are files, while others are goals. The distinction between the two is slight but important. A schema is next presented which allows for the retrieval of all the goals in the system. These operations will always succeed, but could possibly return as its output the emptyset if there are no goals yet in the system.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{GetGoals}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{Manager} \\
\mathit{out}! : \mathbb{P}\, \mathit{NODE} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{out}! = \mathit{goals} \\
\mathit{res}! = \mathit{SUCCESS} \\
\hline
\end{array}$$

In conjunction with the *GetGoals* schema is the *GetFiles* schema. This operation, like the previous one, can return the emptyset if there are no initialized file nodes in the system.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{GetFiles}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{Manager} \\
\mathit{out}! : \mathbb{P}\, \mathit{NODE} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{out}! = \mathit{files} \\
\mathit{res}! = \mathit{SUCCESS} \\
\hline
\end{array}$$

The changing of a node in the system requires searching out all references to that node and changing it. If, for instance, the set *NODE* were names, then this would essentially change the name of a node throughout the system. For this operation to be successful, the node to change must exist already in the system. The new name of the node must also not be a node already in the system. While this latter restriction might be harsh, since it disallows for a user to change two nodes in the tree to the same name, this operation would almost always fail if this were the case. This operation, therefore, is provided for taking one node out of the system and replacing it with another.

The *ChangeNode* operation checks to see if the node to change is a file or a goal, and then changes the node as required. It also accomplishes changing the name throughout the system by converting the old node to the new node in the *commands*, *locations*, *datestamps*, *nodeFiles*, and *nodeGoals* members.

$$
\begin{array}{|l}
\hline
\textit{ChangeNode\_OK} \\
\Delta \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{newNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \in \textit{nodes} \\
\textit{whichNode}? \notin \textit{nodes}' \\
\textit{newNode}? \notin \textit{nodes} \\
\textit{newNode}? \in \textit{nodes}' \\
\textit{whichNode}? \in \textit{goals} \Rightarrow \textit{whichNode}? \notin \textit{goals}' \land \textit{newNode}? \in \textit{goals}' \\
\textit{whichNode}? \in \textit{files} \Rightarrow \textit{whichNode}? \notin \textit{files}' \land \textit{newNode}? \in \textit{files}' \\
\textit{whichNode}? \mapsto x \in \textit{commands} \Rightarrow \textit{whichNode}? \mapsto x \notin \textit{commands}' \land \\
\quad \textit{newNode}? \mapsto x \in \textit{commands}' \\
\textit{whichNode}? \mapsto y \in \textit{locations} \Rightarrow \textit{whichNode}? \mapsto y \notin \textit{locations}' \land \\
\quad \textit{newNode}? \mapsto y \in \textit{locations}' \\
\textit{whichNode}? \mapsto z \in \textit{datestamps} \Rightarrow \textit{whichNode}? \mapsto z \notin \textit{datestamps}' \land \\
\quad \textit{newNode}? \mapsto z \in \textit{datestamps}' \\
\textit{whichNode}? \mapsto a \in \textit{nodeFiles} \Rightarrow \textit{whichNode}? \mapsto a \notin \textit{nodeFiles}' \land \\
\quad \textit{newNode}? \mapsto a \in \textit{nodeFiles}' \\
\forall\, i \mapsto j \in \textit{nodeFiles} \mid \textit{whichNode}? \in j \Rightarrow i \mapsto j \notin \textit{nodeFiles}' \land \\
\quad i \mapsto (j / \textit{whichNode}?) \cup \textit{newNode}? \in \textit{nodeFiles}' \\
\textit{whichNode}? \mapsto b \in \textit{nodeGoals} \Rightarrow \textit{whichNode}? \mapsto b \notin \textit{nodeGoals}' \land \\
\quad \textit{newNode}? \mapsto b \in \textit{nodeGoals}' \\
\forall\, k \mapsto l \in \textit{nodeGoals} \mid \textit{whichNode}? \in k \Rightarrow l \mapsto k \notin \textit{nodeGoals}' \land \\
\quad k \mapsto (l / \textit{whichNode}?) \cup \textit{newNode}? \in \textit{nodeGoals}' \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}
$$

This operation to change an old node into a new node can fail in two ways. One way in which this can happen is if the node to change is not a node in the system at all. It therefore cannot be changed to a new node. The schema which describes this error behavior follows.

$$
\begin{array}{|l}
\hline
\textit{ChangeNode\_ERROR1} \\
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \notin \textit{nodes} \\
\textit{res}! = \textit{FAILURE} \\
\hline
\end{array}
$$

The operation can also fail if the node to change an existing node into is already in the system. Therefore, only new nodes to the system can be the targets of this operation. To complete this operation, the next schema handles this error.

$$
\begin{array}{|l}
\textit{ChangeNode\_ERROR2} \\
\hline
\Xi \textit{Manager} \\
\textit{whichNode?} : \textit{NODE} \\
\textit{newNode?} : \textit{NODE} \\
\textit{res!} : \textit{RESPONSE} \\
\hline
\textit{whichNode?} \in \textit{nodes} \\
\textit{newNode?} \in \textit{nodes} \\
\textit{res!} = \textit{FAILURE} \\
\hline
\end{array}
$$

$$\textit{ChangeNode} \mathrel{\widehat=} \textit{ChangeNode\_OK} \land \textit{ChangeNode\_ERROR1} \land \textit{ChangeNode\_ERROR2}$$

Any node of the system must be able to be deleted. This operation must be complete in its extent and remove all traces of that node throughout the system. All commands associated with that node, the datestamp, location, and children of that node must be removed. The children of the node are not deleted themselves, just the reference that those are the children of the given parent node must be deleted. Then, any parents which refer to the given node must be modified so that they no longer have the given node as a child.

The schema which performs all of these operations is defined next. It requires that the given node to delete be a node in the system for the operation to succeed. Then, it performs all of the deletes that were mentioned above.

$$
\begin{array}{|l}
\textit{DeleteNode\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{aNode?} : \textit{NODE} \\
\textit{res!} : \textit{RESPONSE} \\
\hline
\textit{aNode?} \in \textit{nodes} \\
\textit{aNode?} \notin \textit{nodes}' \\
\textit{aNode?} \in \textit{goals} \Rightarrow \textit{aNode?} \notin \textit{goals}' \\
\textit{aNode?} \in \textit{files} \Rightarrow \textit{aNode?} \notin \textit{files}' \\
\textit{aNode?} \mapsto a \in \textit{commands} \Rightarrow \textit{aNode?} \mapsto a \notin \textit{commands}' \\
\textit{aNode?} \mapsto b \in \textit{locations} \Rightarrow \textit{aNode?} \mapsto b \notin \textit{locations}' \\
\textit{aNode?} \mapsto c \in \textit{datestamps} \Rightarrow \textit{aNode?} \mapsto c \notin \textit{datestamps}' \\
\textit{aNode?} \mapsto n \in \textit{nodeGoals} \\
n \neq \varnothing \Rightarrow \textit{aNode?} \mapsto n \notin \textit{nodeGoals}' \land \textit{aNode?} \mapsto \varnothing \in \textit{nodeGoals}' \\
\forall\, w \mapsto x \in \textit{nodeGoals} \mid \textit{aNode?} \in x \Rightarrow w \mapsto x \notin \textit{nodeGoals}' \land w \mapsto x/\textit{aNode?} \in \textit{nodeGoals}' \\
\textit{aNode?} \mapsto m \in \textit{nodeFiles} \\
m \neq \varnothing \Rightarrow \textit{aNode?} \mapsto m \notin \textit{nodeFiles}' \land \textit{aNode?} \mapsto \varnothing \in \textit{nodeFiles}' \\
\forall\, y \mapsto z \in \textit{nodeFiles} \mid \textit{aNode?} \in z \Rightarrow y \mapsto z \notin \textit{nodeFiles}' \land y \mapsto z/\textit{aNode?} \in \textit{nodeFiles}' \\
\textit{res!} = \textit{SUCCESS} \\
\hline
\end{array}
$$

This operation can fail if the given node is not a node in the system. The schema which describes this behavior follows.

```
DeleteNode_ERROR
 ΞManager
 aNode? : NODE
 res! : RESPONSE
 ──────────
 aNode? ∉ nodes
 res! = FAILURE
```

$DeleteNode \widehat{=} DeleteNode\_OK \land DeleteNode\_ERROR$

4.4.4 Node Schemas

The vast majority of operations in the system are performed on a specific node. These allow for the changing and querying of all the nodes in the system. Some of these nodes are goals, while some are files.

The *AddNodeGoal* and *AddNodeFile* operations add children to a specific node and initialize these new nodes in the system. *GetNodesGoals* and *GetNodesFiles* provide ways to get the sets of children of a node. These are vital in any operation that traverses the dependency tree such as the compile operation or the displaying of the tree graphically to the user. The *GetNodesCommands* schema allows for the retrieval of the commands of a specific goal. This is also vital in determining what operations to execute should it be deemed necessary. Similarly, the *ChangeCommand* and *DeleteCommand* operations bring about change in the list of commands associated with a node. Finally, the *DeleteNodeNode* schema provides a way to break the line between two nodes. It does not actually delete the target node, as it possibly has multiple parents.

These operations are defined more concisely next.

There is a specific date named *NEW_DATE*. This date is defined to be so early in time that reasonably any date when compared to this date will be newer than this date. For instance, *January 1, 0000* would be a good choice or on UNIX systems, the 32-bit date of all zeros which would be the beginning of the UNIX clock would be another good choice. There is also a specific location given as the default to all files named *NEW_LOCATION*. A reasonable choice for this location would be the current directory on the local machine. This is used to insure that every file has a location, even if the user does not specify it explicitly.

There must be an operation to add a file as a child of a specified node. This operation requires that the input node be a node in the system. It also has the ability to add the input file node to the system if it is not already in the system. This is important because there should not be duplicates of any node; so if this node is becoming a child of its second parent, it is already in the system. To accommodate this, a method is provided for placing a node into the system without requiring a separate schema. It also ensures that the file that is to be added will have a parent, and therefore will not be *Top Level*. The children of this new file, should it be a new file to the system, are set to the emptyset as well. A check is performed that disallows a node from placing multiple dependencies on the same file. In this way, the file node to add cannot already be a child of this node. If the file to add has no datestamp it is given the date defined above, *NEW_DATE*. Similarly, if the file defined above has no location, it is given the location defined above, *NEW_LOCATION*. Finally, the operation places the file as a child file of the given node.

$$
\begin{array}{|l}
\textit{AddNodeFile\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{aNode}? : \textit{NODE} \\
\textit{aFile}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{aNode}? \in \textit{nodes} \\
\textit{aFile}? \notin \textit{nodes} \Rightarrow \textit{aFile}? \in \textit{nodes}' \\
\textit{aFile}? \notin \textit{files} \Rightarrow \textit{aFile}? \in \textit{files}' \land \textit{aFile}? \mapsto \varnothing \in \textit{nodeGoals}' \land \\
\qquad \textit{aFile}? \mapsto \varnothing \in \textit{nodeFiles}' \\
\textit{aNode}? \mapsto b \in \textit{nodeFiles} \Rightarrow \textit{aFile}? \notin b \\
\textit{aFile}? \mapsto x \notin \textit{datestamps} \Rightarrow \textit{aFile}? \mapsto \textit{NEW\_DATE} \in \textit{datestamps}' \\
\textit{aFile}? \mapsto y \notin \textit{locations} \Rightarrow \textit{aFile}? \mapsto \textit{NEW\_LOCATION} \in \textit{locations}' \\
\textit{aNode}? \mapsto z \in \textit{nodeFiles} \Rightarrow \textit{aNode}? \mapsto z \notin \textit{nodeFiles}' \land \textit{aNode}? \mapsto z \cup \textit{aFile}? \in \textit{nodeFiles}' \\
\textit{aNode}? \mapsto a \notin \textit{nodeFiles} \Rightarrow \textit{aNode}? \mapsto \textit{aFile}? \in \textit{nodeFiles}' \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}
$$

One way this operation could fail is simply if one attempts to add a child to a nonexistent node. It therefore cannot have children until it has been created either through a top level goal or through performing an *AddNodeGoal* on another node.

$$
\begin{array}{|l}
\textit{AddNodeFile\_ERROR1} \\
\hline
\Xi \textit{Manager} \\
\textit{aNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{aNode}? \notin \textit{nodes} \\
\textit{res}! = \textit{FAILURE} \\
\hline
\end{array}
$$

The last way the operation could fail is if the node is already a child; adding the node as a child again would create duplicate children. This ensures it is ensured that any node will not have multiple dependencies on the same file.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AddNodeFile\_ERROR2 \qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Manager \\
aNode? : NODE \\
aFile? : NODE \\
res! : RESPONSE \\
\hline
aNode? \in nodes \\
aNode? \mapsto b \in nodeFiles \Rightarrow aFile? \in b \\
res! = FAILURE \\
\hline
\end{array}$$

$$AddNodeFile \mathrel{\widehat=} AddNodeFile\_OK \land AddNodeFile\_ERROR1 \land AddNodeFile\_ERROR2$$

Similar to the *AddNodeFile* operation outlined above, this operation adds a goal child to a node. It requires that the input node, to which a goal will be added to, already be in the system. If the goal to add as a child of the given node is not in the system already, this operation will create the goal. Unlike the *AddNodeFile* operation defined above, the goal child for this operation does not get the *NEW_DATE* and *NEW_LOCATION* stamps defined for it. This is because goals do not have dates or locations.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AddNodeGoal\_OK \qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Manager \\
aNode? : NODE \\
aGoal? : NODE \\
res! : RESPONSE \\
\hline
aNode? \in nodes \\
aGoal? \notin nodes \Rightarrow aGoal? \in nodes' \\
aGoal? \notin goals \Rightarrow aGoal? \in goals' \land aGoal? \mapsto \varnothing \in nodeGoals' \land \\
\qquad aGoal? \mapsto \varnothing \in nodeFiles' \\
aNode? \mapsto b \in nodeGoals \Rightarrow aGoal? \notin b \\
aNode? \mapsto z \in nodeGoals \Rightarrow aNode? \mapsto z \notin nodeGoals' \land aNode? \mapsto z \cup Goal? \in nodeGoals' \\
aNode? \mapsto a \notin nodeGoals \Rightarrow aNode? \mapsto aGoal? \in nodeGoals' \\
res! = SUCCESS \\
\hline
\end{array}$$

One way this operation can fail is if the input node is not already in the system. The schema which describes this operation follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AddNodeGoal\_ERROR1 \qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Manager \\
aNode? : NODE \\
res! : RESPONSE \\
\hline
aNode? \notin nodes \\
res! = FAILURE \\
\hline
\end{array}$$

The other way this operation can fail is if the input goal is already a goal of the specified node. In this way it is disallowed for a node to depend on the same goal twice. To handle this other error case, this next schema is defined.

$AddNodeGoal\_ERROR2$

$$\begin{array}{|l}
\Xi Manager \\
aNode? : NODE \\
aGoal? : NODE \\
res! : RESPONSE \\
\hline
aNode? \in nodes \\
aNode? \mapsto b \in nodeGoals \Rightarrow aGoal? \in b \\
res! = FAILURE
\end{array}$$

$$AddNodeGoal \mathrel{\widehat=} AddNodeGoal\_OK \land AddNodeGoal\_ERROR1 \land AddNodeGoal\_ERROR2$$

Each node can have as its children a set of goals and / or a set of files. Therefore it is necessary to allow for the retrieval of the children goals of a particular node. The input node must be a node in the system and if so, the child goals of that node are returned. Should a node have no child goals, an emptyset is used as output.

$GetNodesGoals\_OK$

$$\begin{array}{|l}
\Xi Manager \\
whichNode? : NODE \\
out! : \mathbb{P}\, NODE \\
res! : RESPONSE \\
\hline
whichNode? \in nodes \\
whichNode? \mapsto x \in nodeGoals \Rightarrow out! = x \\
whichNode? \mapsto x \notin nodeGoals \Rightarrow out! = \varnothing \\
res! = SUCCESS
\end{array}$$

If the requested node is not a node in the system, then the operation to get a node's children goals must fail. An operation is next defined which handles this error condition.

$GetNodesGoals\_ERROR$

$$\begin{array}{|l}
\Xi Manager \\
whichNode? : NODE \\
res! : RESPONSE \\
\hline
whichNode? \notin nodes \\
res! = FAILURE
\end{array}$$

$$GetNodesGoals \mathrel{\widehat=} GetNodesGoals\_OK \land GetNodesGoals\_ERROR$$

Similarly, the operation to get the child files of a particular node must be defined. The schema takes as input a node already existing in the system and has for its output the files that are children of that node. Again, if a particular node has no children that are files, the emptyset is returned.

$$
\begin{array}{|l}
\textit{GetNodesFiles\_OK} \\
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{out}! : \mathbb{P}\, \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \in \textit{nodes} \\
\textit{whichNode}? \mapsto x \in \textit{nodeFiles} \Rightarrow \textit{out}! = x \\
\textit{whichNode}? \mapsto x \notin \textit{nodeFiles} \Rightarrow \textit{out}! = \emptyset \\
\textit{res}! = \textit{SUCCESS}
\end{array}
$$

If the input node is not contained in the system the operation must fail. This error condition is handled with the next schema.

$$
\begin{array}{|l}
\textit{GetNodesFiles\_ERROR} \\
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \notin \textit{nodes} \\
\textit{res}! = \textit{FAILURE}
\end{array}
$$

$$\textit{GetNodesFiles} \mathrel{\widehat=} \textit{GetNodesFiles\_OK} \wedge \textit{GetNodesFiles\_ERROR}$$

To get the command of a node, the node requested must first be a goal contained in the system. This is because only goals can have commands, where files cannot. The output is in the form of a set of commands. This is done so as to allow for multiple commands to be issued at a goal should it need to be updated.

$$
\begin{array}{|l}
\textit{GetNodesCommands\_OK} \\
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{out}! : \mathbb{P}\, \textit{COMMAND} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \in \textit{goals} \\
\textit{whichNode}? \mapsto x \in \textit{commands} \Rightarrow \textit{out}! = x \\
\textit{whichNode}? \mapsto x \notin \textit{commands} \Rightarrow \textit{out}! = \emptyset \\
\textit{res}! = \textit{SUCCESS}
\end{array}
$$

If the node requested is not a goal in the system, then there will be an error. In this way the schema helps to ensure that only goals have commands associated with them.

$$\begin{array}{|l}
\textit{GetNodesCommands\_ERROR} \\
\hline
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \notin \textit{goals} \\
\textit{res}! = \textit{FAILURE} \\
\hline
\end{array}$$

$$\textit{GetNodesCommands} \mathrel{\widehat{=}} \textit{GetNodesCommands\_OK} \land \textit{GetNodesCommands\_ERROR}$$

Only goals can have commands associated with them. These commands are executed when the goal must be updated. In a typical example, these commands might be to compile the children of a goal into some object file, perhaps move that object file to a class directory, and then delete it from the current directory.

The operation to change the commands of a given node is specified next. It requires that the node to change be a goal in the system. If the goal already has associated with it a set of commands, then these commands are unbound from that goal. Finally, the new set of commands gets bound to the goal.

$$\begin{array}{|l}
\textit{ChangeCommand\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{newCommands}? : \mathbb{P}\, \textit{COMMAND} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \in \textit{goals} \\
\textit{whichNode}? \mapsto a \in \textit{commands} \Rightarrow \textit{whichNode}? \mapsto a \notin \textit{commands}' \\
\textit{whichNode}? \mapsto \textit{newCommands}? \in \textit{commands}' \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}$$

A proof of the *ChangeCommand_OK* schema is provided next. This is used to show that the data invariant is not violated. Here it is necessary to show that *whichNode?* maps to both any arbitrary command, as well as the new command, is valid only if *whichNode?* is a goal. This is again necessary because only goals can have commands associated with them.

(1)	$\textit{whichNode}? \in \textit{NODE}$	(premise)
(2)	$\textit{newCommands}? \subset \mathbb{P}\, \textit{COMMAND}$	(premise)
(3)	$\textit{whichNode}? \in \textit{goals}$	(premise)
(4)	$\textit{whichNode}? \mapsto a \in \textit{commands} \Rightarrow \textit{whichNode}? \mapsto a \notin \textit{commands}'$	(premise)
(5)	$\textit{whichNode}? \mapsto \textit{newCommands}? \in \textit{commands}'$	(premise)
(6)	$\textit{whichNode}? \mapsto a \in \textit{commands} \Rightarrow \textit{whichNode}? \in \textit{goals}$	((3), (4), subst.)
(7)	$\textit{whichNode}? \mapsto \textit{newCommands}? \in \textit{commands}' \Rightarrow \textit{whichNode}? \in \textit{goals}$	((3), (5), subst.)

Q.E.D.

If the given node to change the commands associated with it is not a goal in the system, then the operation must fail. This helps to ensure that only goals have commands associated with them.

$$
\begin{array}{|l}
\textit{ChangeCommand\_ERROR} \\
\hline
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \notin \textit{goals} \\
\textit{res}! = \textit{FAILURE} \\
\hline
\end{array}
$$

$$\textit{ChangeCommand} \mathrel{\widehat{=}} \textit{ChangeCommand\_OK} \land \textit{ChangeCommand\_ERROR}$$

Each node has associated with it children nodes. These nodes can be either goals or files, but they are nevertheless, still nodes. The operation to delete one of the node children of a specified node does not delete the node from the system. It merely severs the parental link, it disowns the specified node. The node still exists in the system. It is important to note that a node that has a few parents can be disowned by all parents and then suddenly become a node with no parental links but still exist in the system. This is recognized and not prevented for several reasons. One reason is that a user might wish to disconnect a node from the system temporarily. Perhaps it is believed that a particular node is causing problems for the program and this is one means that the user feels will help bug testing. Another reason is that maybe a file is no longer needed for this particular version of the product but it may be needed to recall an earlier version.

The operation to sever a dependency of a node upon another node is defined next. It requires that the parent node be in the system and that the child node specified really be a child of that parent. If these are both the case, then the association between the two files is removed.

$$
\begin{array}{|l}
\textit{DeleteNodeNode\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{delNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \in \textit{nodes} \\
\textit{delNode}? \in \textit{nodes} \\
\textit{whichNode}? \mapsto x \in \textit{nodeGoals} \land \textit{delNode}? \in x \lor \textit{whichNode}? \mapsto y \in \textit{nodeFiles} \land \textit{delNode}? \in y \\
\textit{whichNode}? \mapsto a \in \textit{nodeGoals} \land \textit{delNode}? \in x \Rightarrow \textit{whichNode}? \mapsto a \notin \textit{nodeGoals}' \land \\
\qquad \textit{whichNode}? \mapsto a / \textit{delNode}? \in \textit{nodeGoals}' \\
\textit{whichNode}? \mapsto b \in \textit{nodeFiles} \land \textit{delNode}? \in x \Rightarrow \textit{whichNode}? \mapsto b \notin \textit{nodeFiles}' \land \\
\qquad \textit{whichNode}? \mapsto b / \textit{delNode}? \in \textit{nodeFiles}' \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}
$$

This operation can fail in two ways. One is if the parent node is not a node in the system. The schema which outlines this behavior follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DeleteNodeNode\_ERROR}1} \\
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \notin \textit{nodes} \\
\textit{res}! = \textit{FAILURE}
\end{array}$$

The other way in which this operation can fail is if the given parental node does not have as a child of it the given child node. It, therefore, cannot be severed and the operation must fail. The schema which outlines this follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DeleteNodeNode\_ERROR}2} \\
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{delNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \in \textit{nodes} \\
\textit{whichNode}? \mapsto x \in \textit{nodeGoals} \land \textit{delNode}? \notin x \lor \textit{whichNode}? \mapsto y \in \textit{nodeFiles} \land \textit{delNode}? \notin y \\
\textit{res}! = \textit{FAILURE}
\end{array}$$

$$\textit{DeleteNodeNode} \mathrel{\widehat=} \textit{DeleteNodeNode\_OK} \land \textit{DeleteNodeNode\_ERROR}1 \land \textit{DeleteNodeNode\_ERROR}2$$

There must be an operation to delete one of the commands associated with a specified node. Since these commands can only be inserted through the *ChangeCommand* schema which itself makes sure that the node is a goal, that check is not needed here. This operation takes a command of a node and removes it from the list of commands associated with a node. It requires that the given node be a node that is bound to a list of commands already, and that the given command be one of those commands. The operation is defined next.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DeleteCommand\_OK}} \\
\Delta \textit{Manager} \\
\textit{aNode}? : \textit{NODE} \\
\textit{aCommand}? : \textit{COMMAND} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{aNode}? \mapsto a \in \textit{commands} \\
\textit{aCommand}? \in a \\
\textit{aNode}? \mapsto a \notin \textit{commands}' \\
\textit{aNode}? \mapsto a / \textit{aCommand}? \in \textit{commands}' \\
\textit{res}! = \textit{SUCCESS}
\end{array}$$

This operation can fail in two ways. One way is if the given node has no commands associated with it at all. The schema which follows outlines this behavior.

$DeleteCommand\_ERROR1$

$\Xi Manager$
$aNode? : NODE$
$res! : RESPONSE$

$aNode? \mapsto a \notin commands$
$res! = FAILURE$

The other way in which a node can fail is if the given node does not have the given command associated with it. Next is the schema which describes this operation.

$DeleteCommand\_ERROR2$

$\Xi Manager$
$aNode? : NODE$
$aCommand? : COMMAND$
$res! : RESPONSE$

$aNode? \mapsto a \in commands$
$aCommand? \notin a$
$res! = FAILURE$

$$DeleteCommands \mathrel{\widehat=} DeleteCommand\_OK \land DeleteCommand\_ERROR1 \land DeleteCommand\_ERROR2$$

4.4.5 File Schemas

Files are different from goals in several ways. One is that they actually exist in the network and, as such, have a location. Also, because of this existence, they have dates associated with them when they were last modified. These properties of a file become important when determining whether a file has changed and needs to be recompiled and if so, where to get it.

Operations must be provided to allow for interaction with these properties of a file. The *AddLogin* schema provides a way to bind a login name and password to a specific URL. This is very useful when a file's location is on a non-anonymous FTP site. *GetFileDate* and *GetFileLocation* provide simple queries which return what the system believes the last modified date and the location of the file, respectively. Another operation must be employed to get the most current date of a file and check it against the date provided by *GetFileDate*. The *ChangeLogin* operation is provided to change the login name and password bound to a particular URL. *ChangeDate* is a very special operation which must be protected in the final implementation. This operation allows for telling the system the date that a file was last modified. The system then must update itself accordingly. This should be called during a compilation when it is determined that a file's actual date is newer than the date the system has recorded. Finally, *ChangeLocation* and *DeleteLogin* provide ways to inform the system that a file has moved to a new location and delete a login associated with a URL, respectively.

These operations are defined in more detail next.

Some URLs require passwords and login names such as non-anonymous FTP sites. For this, an operation is provided to bind a login name and a password to a URL. To succeed, the input URL must be defined as a location of one of the files already in the system. The URL must not already have a login and password associated with it. If that is the case, the *ChangeLogin* or *DeleteLogin* operations should be used.

$$
\begin{array}{|l}
\textit{AddLogin\_OK} \\
\hline
\Delta \textit{Manager} \\
aURL? : URL \\
aLogin? : LOGIN \\
aPass? : PASSWORD \\
res! : RESPONSE \\
\hline
a \mapsto aURL? \in locations \\
aURL? \mapsto b \notin logins \\
logins' = logins \cup (aURL?, (aLogin?, aPass?)) \\
res! = SUCCESS \\
\hline
\end{array}
$$

The above schema is proven next. Here it is shown that the input URL can only be linked to a login and password if that URL is already bound to a location.

(1)	$aURL? \in URL$	(premise)
(2)	$aLogin? \in LOGIN$	(premise)
(3)	$aPass? \in PASSWORD$	(premise)
(4)	$a \mapsto aURL? \in locations$	(premise)
(5)	$logins' = logins \cup (aURL?, (aLogin?, aPass?))$	(premise)
(6)	$aURL? \mapsto (aLogin?, aPass?) \in logins \Rightarrow a \mapsto aURL? \in locations$	((4), (5), subst.)

Q.E.D.

If the given URL is not bound to a file in the system, then this operation fails. In this way it is disallowed for a user to bind a login name and password to a random URL without that URL having a distinct purpose in the system.

$$
\begin{array}{|l}
\hline
\mathit{AddLogin\_ERROR}1 \\
\Xi \mathit{Manager} \\
\mathit{aURL}? : \mathit{URL} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
a \mapsto \mathit{aURL}? \notin \mathit{locations} \\
\mathit{res}! = \mathit{FAILURE} \\
\hline
\end{array}
$$

The operation can also fail if the given URL already has a mapping to a login name and a password. This is handled with the following schema.

$$
\begin{array}{|l}
\hline
\mathit{AddLogin\_ERROR}2 \\
\Xi \mathit{Manager} \\
\mathit{aURL}? : \mathit{URL} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aURL}? \mapsto a \in \mathit{logins} \\
\mathit{res}! = \mathit{FAILURE} \\
\hline
\end{array}
$$

$$
\mathit{AddLogin} \mathrel{\widehat{=}} \mathit{AddLogin\_OK} \wedge \mathit{AddLogin\_ERROR}1 \wedge \mathit{AddLogin\_ERROR}2
$$

Each file has associated with it a date when it was last updated in the system. A file has been updated if it has changed since the time the system has recorded it had changed. Also, even if a file has not changed, it still is marked as updated if any of its children are updated. In this way, the property that a branch of a tree has been updated is obtained. A file will at all times have some date associated with it, even before its first update. This operation takes a file node as input and returns the date. The operation requires that the node passed as input be an existing file in the system to succeed. For more information on how datestamps work, see Section 4.6.

$$
\begin{array}{|l}
\hline
\mathit{GetFileDate\_OK} \\
\Xi \mathit{Manager} \\
\mathit{aNode}? : \mathit{NODE} \\
\mathit{aDate}! : \mathit{Date} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aNode}? \in \mathit{files} \\
\mathit{aNode}? \mapsto x \in \mathit{datestamps} \Rightarrow \mathit{aDate}! = x \\
\mathit{res}! = \mathit{SUCCESS} \\
\hline
\end{array}
$$

Should the input node *not* be a file node in the system, the operation must fail. This is one way in which it is maintained that only file nodes and not goal nodes have dates associated with them.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{GetFileDate\_ERROR}} \\
\Xi \mathit{Manager} \\
\mathit{aNode}? : \mathit{NODE} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aNode}? \notin \mathit{files} \\
\mathit{res}! = \mathit{FAILURE} \\
\hline
\end{array}$$

$$\mathit{GetFileDate} \mathrel{\widehat=} \mathit{GetFileDate\_OK} \land \mathit{GetFileDate\_ERROR}$$

All files have a location associated with them. In some cases this is through the Hyper Text Transfer Protocol (HTTP) or File Transfer Protocol (FTP). More commonly, however, the files are likely to be located locally. This is achieved through the FILE protocol. In this way, all file locations can be represented as a Uniform Resource Locator (URL).

It is, therefore, important that the location of a file can be queried. This operation requires that the input node be a file in the system and sets as its output the location of that file.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{GetFileLocation\_OK}} \\
\Xi \mathit{Manager} \\
\mathit{aNode}? : \mathit{NODE} \\
\mathit{aLocation}! : \mathit{URL} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aNode}? \in \mathit{files} \\
\mathit{aNode}? \mapsto x \in \mathit{locations} \Rightarrow \mathit{aLocation}! = x \\
\mathit{res}! = \mathit{SUCCESS} \\
\hline
\end{array}$$

The operation can fail should the input node not be contained in the set of files that the system recognizes. The schema that handles this error condition is defined next.

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{GetFileLocation\_ERROR}} \\
\Xi \mathit{Manager} \\
\mathit{aNode}? : \mathit{NODE} \\
\mathit{res}! : \mathit{RESPONSE} \\
\hline
\mathit{aNode}? \notin \mathit{files} \\
\mathit{res}! = \mathit{FAILURE} \\
\hline
\end{array}$$

$$\mathit{GetFileLocation} \mathrel{\widehat=} \mathit{GetFileLocation\_OK} \land \mathit{GetFileLocation\_ERROR}$$

While it might not be likely that a user needs to change the login name or password associated with a URL, the operation to allow for this must be provided. The schema shown below requires that the given URL to change be bound to a login name and password already. This operation then removes that bind, and rebinds the input login and password to the input URL.

$$\begin{array}{|l}
\textit{ChangeLogin\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{aURL}? : \textit{URL} \\
\textit{aLogin}? : \textit{LOGIN} \\
\textit{aPass}? : \textit{PASSWORD} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{aURL}? \mapsto b \in \textit{logins} \\
\textit{aURL}? \mapsto b \notin \textit{logins}' \\
\textit{aURL}? \mapsto (\textit{aLogin}?, \textit{aPass}?) \in \textit{logins}' \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}$$

This operation could fail if the specified URL is not already bound to a login and password. If this is the case, the *AddLogin* operation should be used. The *AddLogin* operation checks to be certain that the URL to bind a login and password to is itself bound to a location. That check is not performed here because to change a login, it must already be bound to a URL.

$$\begin{array}{|l}
\textit{ChangeLogin\_ERROR} \\
\hline
\Xi \textit{Manager} \\
\textit{aURL}? : \textit{URL} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{aURL}? \mapsto b \notin \textit{logins} \\
\textit{res}! = \textit{FAILURE} \\
\hline
\end{array}$$

$$\textit{ChangeLogin} \mathrel{\widehat=} \textit{ChangeLogin\_OK} \land \textit{ChangeLogin\_ERROR}$$

Various changes can be made to the attributes of a node after its initial creation. An obvious change that will frequently occur is the changing of the datestamp associated with a file. Every time a user updates a file, the datestamp will require updating. The *ChangeDate* function that will accomplish this task should allow the user no means of manually altering the datestamp associated a file. For instance, there is no reason why the user would be given an option to change the date a file was last modified like they will be given the option to change the location of a file. This operation, therefore, must be protected in the implementation so that it is only accessible by the compile operation. This operation will deem a file's date has changed by going through the POSIX interface and checking a file's last modified date.

The following schema describes the system function for updating a datestamp. It requires that the node to change be a file already in the system. In this way, it is ensured that goals do not have datestamps associated with them, since this is not allowed.

ChangeDate_OK

$\Delta Manager$
$whichNode? : NODE$
$newDate? : DATE$
$res! : RESPONSE$

$whichNode? \in files$
$whichNode? \mapsto newDate? \in datestamps'$
$whichNode? \mapsto x \in datestamps' \Rightarrow |x| = 1$
$res! = SUCCESS$

This operation will fail if invalid fields are entered for the new datestamp, if the node does not yet exist, or if the node refers to a goal rather than a file. In addition, the schema verifies that there is only one datestamp associated with each file.

ChangeDate_ERROR

$\Xi Manager$
$whichNode? : NODE$
$res! : RESPONSE$

$whichNode? \notin files$
$res! = FAILURE$

$$ChangeDate \widehat{=} ChangeDate\_OK \wedge ChangeDate\_ERROR$$

Files have locations, either locally or throughout the network. Once the user has entered the location of one of those files, however, it is feasible that the location might change at some time. Perhaps the file is moved to a different directory or to a different site. Therefore, an operation must be provided to change the pointer to the location of a file.

This operation is outlined below and requires that the input node be a file. If the file node in which to change the location of already has a location, then it is changed over to the new location. However, if it does not have a location, then it is given one. This should never occur since every node is given a default location, as described in Section 4.4.4, and there is no operation to delete the location of a file. Any file that does not have a location, is clearly no longer a file or no longer needs to be in the system and must be deleted through other means such as *DeleteNode*.

$$
\begin{array}{|l}
\textit{ChangeLoc\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{newLoc}? : \textit{URL} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichnode} \in \textit{files} \\
\textit{whichNode}? \mapsto \textit{newLoc}? \in \textit{locations}' \\
\textit{whichNode}? \mapsto x \in \textit{locations}' \Rightarrow |x| = 1 \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}
$$

If the user specifies a node to change that is not one of the files that currently exists in the system, this operation must fail. This helps to ensure that only files have locations associated with them.

$$
\begin{array}{|l}
\textit{ChangeLoc\_ERROR} \\
\hline
\Xi \textit{Manager} \\
\textit{whichNode}? : \textit{NODE} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{whichNode}? \notin \textit{files} \\
\textit{res}! = \textit{FAILURE} \\
\hline
\end{array}
$$

$$\textit{ChangeLoc} \mathrel{\widehat=} \textit{ChangeLoc\_OK} \land \textit{ChangeLoc\_ERROR}$$

Once a login name and password have been bound to a URL there must be an operation available to unbind them. This operation could be used, albeit seldomly, when a non-anonymous FTP site changes to an anonymous FTP site and the user wants to delete the login and password references that had previously been defined.

The operation to delete a login associated with a specific URL is specified next. It requires that the URL to unbind be actually bound to a login already. Since the operation to actually bind a login and password to a URL requires itself that the URL be a valid URL in the system, that check is not needed here.

$$
\begin{array}{|l}
\textit{DeleteLogin\_OK} \\
\hline
\Delta \textit{Manager} \\
\textit{aURL}? : \textit{URL} \\
\textit{res}! : \textit{RESPONSE} \\
\hline
\textit{aURL}? \mapsto b \in \textit{logins} \\
\textit{aURL}? \mapsto b \notin \textit{logins}' \\
\textit{res}! = \textit{SUCCESS} \\
\hline
\end{array}
$$

If, however, the given URL does not already have a login and password associated with it, then this operation must fail. The schema which specifies this behavior follows.

$$
\begin{array}{|l}
\hline
\textit{DeleteLogin\_ERROR} \\
\Xi Manager \\
aURL? : URL \\
res! : RESPONSE \\
\hline
aURL? \mapsto b \notin logins \\
res! = FAILURE \\
\hline
\end{array}
$$

$DeleteLogin \mathrel{\widehat=} DeleteLogin\_OK \land DeleteLogin\_ERROR$

4.4.6 Compilation Specification

Possibly the most important operation of the system is the compilation operation. This is similar to the *Make* operation. However, in the system described here, the dependency tree will be traversed and all the nodes hit. This will allow for a compile of the entire project that is currently loaded.

When a node *hits* another node, it activates that node. The activator node then awaits all its children to be completed. At that point it can complete its operation. To provide for a way for nodes with multiple parents not to be activated multiple times, a type of semaphore was contrived. This is accomplished through a state variable that can take one of four states.

As mentioned in Section 4.4.2, the state is a variable which can be either *BUSY*, *NOT_HIT*, *CHANGED*, or *NOT_CHANGED*. The variable is initialized when a compile operation is begun and at that time all nodes are set to *NOT_HIT*. As nodes use their children to determine if they must recompile, the state of their children nodes becomes very important. Therefore, there must be an operation to get the state of a particular node. The following schema returns the state of a node if that node currently has a state assigned to it. A node could have no state at all if the system is not performing a compile operation. This is done so that the *states* member will not be holding information during the time when a user is merely changing or entering data.

$$
\begin{array}{|l}
\hline
\textit{GetState\_OK} \\
\Xi Manager \\
aNode? : NODE \\
state! : STATE \\
res! : RESPONSE \\
\hline
aNode? \in nodes \\
aNode? \mapsto x \in states \\
state! = x \\
res! = SUCCESS \\
\hline
\end{array}
$$

The operation to get the state of a node can fail if the input node is not a node recognized by the system. The schema that outlines this error is presented next.

GetState_ERROR

$\Xi Manager$
$aNode? : NODE$
$res! : RESPONSE$

$aNode? \notin nodes$
$res! = FAILURE$

$GetState \mathrel{\widehat{=}} GetState\_OK \wedge GetState\_ERROR$

During the compilation operation, nodes will be changing their states upon successfully being activated and updated. This is explained further in the section on compiling. There must, therefore, be an operation to change the state of a particular node. This operation must take as its input a valid node in the system to be successful. Then, it changes the state associated with that node to the new state specified as input.

ChangeState_OK

$\Delta Manager$
$whichNode? : NODE$
$newState? : STATE$
$res! : RESPONSE$

$whichNode? \in nodes$
$whichNode? \mapsto x \in states \Rightarrow whichNode? \mapsto x \notin states'$
$whichNode? \mapsto newState? \in states'$
$res! : SUCCESS$

A proof of the *ChangeState_OK* schema follows which proves that the changing of a state variable associated with a node does not violate the data invariant of the *Manager*.

(1)	$whichNode? \in nodes$	(premise)
(2)	$whichNode? \mapsto x \in states \Rightarrow whichNode? \mapsto x \notin states'$	(premise)
(3)	$whichNode? \mapsto newState? \in states'$	(premise)
(4)	$whichNode? \mapsto states \Rightarrow whichNode? \in nodes$	((1), subst.)

Q.E.D.

The operation will fail if the given node is not an existing node in the system already. The schema that handles this error is presented next.

ChangeState_ERROR

$\Xi Manager$
$whichNode? : NODE$
$res! : RESPONSE$

$whichNode? \notin nodes$
$res! : FAILURE$

$$ChangeState \mathrel{\widehat=} ChangeState\_OK \land ChangeState\_ERROR$$

The above two schemas allow for interaction with the state of a node. When a node initializes all of its children, it becomes busy and then waits for all of its children to no longer be busy. This algorithm is explained next.

The algorithm for executing the nodes' commands in order is important in the configuration manager. This method is efficient and uses recursion to cover all of the nodes in the execution system. It also allows for parallelism so that each node can run in parallel and communicate through its state, or semaphore described earlier.

The algorithm is quite straightforward. The user can pick any goal from which to begin compilation. This goal becomes the compilation root. When the user begins compilation, the states are initialized. This sets the semaphore for all the goals to a *NOT_HIT* state. The schema which describes the initialization of the compilation operation is provided next.

Init_Compile

$\Delta Manager$

$\forall\, a \in nodes \mid a \mapsto NOT\_HIT \in states'$

Next the compilation root sets its state to *BUSY* and enters the *Start_Spawn* operation. For each child with a state of *NOT_HIT*, a thread is spawned for that child with the spawnThread(node) operation. This thread will treat the child as a new compilation root from which the process starts again. The operation that checks the state of the child and spawns a new thread must be atomic, due to multiple dependencies on any give node. This will prevent multiple threads being spawned for a single child with multiple parents. The node will continue to check the state of it's children with the *Is_Busy* operation. While in this state, all the children of a node are determining if they have changed or any of their children have changed. Upon completion, the *BUSY* state will change to either *CHANGED* or *NOT_CHANGED.* When the state of all of the node's children is *CHANGED* or *NOT_CHANGED*, the node will then enter the *End_Spawn* operation and update it's own state accordingly. At this point, a node evaluates if its children have changed. If they have, then it must also set its state to *CHANGED.* However, if they have not, but it is a file and has physically been changed since the last date recorded in the system, it must also set its state to *CHANGED*. If a node has changed and is a goal, it will then execute any commands with the *Execute_Commands* operation.

The next schema is used when a node is first hit. It sets its state to *BUSY* and then spawns all of its children.

$$\begin{array}{|l}
\textit{Start\_Spawn} \\
\Delta \mathit{Manager} \\
\mathit{whichnode?} : \mathit{NODE} \\
\hline
\mathit{whichnode?} \mapsto \mathit{NOT\_HIT} \in \mathit{states} \\
\mathit{whichnode?} \mapsto \mathit{BUSY} \in \mathit{states}' \\
\mathit{whichnode?} \mapsto a \in \mathit{nodeGoals} \Rightarrow \forall\, b \in a \mid b \mapsto \mathit{NOT\_HIT} \in \mathit{states} \Rightarrow \mathit{spawnThread}(b) \\
\mathit{whichnode?} \mapsto c \in \mathit{nodeFiles} \Rightarrow \forall\, d \in c \mid d \mapsto \mathit{NOT\_HIT} \in \mathit{states} \Rightarrow \mathit{spawnThread}(d) \\
\hline
\end{array}$$

After spawning all of its children, a node then goes into a wait state where it waits for its children to come out of busy. Next, the operation where at least one of a node's children is still busy is defined.

$$\begin{array}{|l}
\textit{Busy\_Yes} \\
\Xi \mathit{Manager} \\
\mathit{whichnode?} : \mathit{NODE} \\
\mathit{result!} : \mathit{YES} \mid \mathit{NO} \\
\hline
\mathit{whichnode?} \mapsto b \in \mathit{nodeGoals} \Rightarrow \exists\, c \in b \mid c \mapsto \mathit{BUSY} \in \mathit{states} \\
\mathit{whichnode?} \mapsto d \in \mathit{nodeFiles} \Rightarrow \exists\, e \in d \mid e \mapsto \mathit{BUSY} \in \mathit{states} \\
\mathit{result!} = \mathit{YES} \\
\hline
\end{array}$$

The next operation is used when all of the children of a node have come out of the *BUSY* state. At this point, a node needs to call the *End_Spawn* operation.

$$\begin{array}{|l}
\textit{Busy\_No} \\
\Xi \mathit{Manager} \\
\mathit{whichnode?} : \mathit{NODE} \\
\mathit{result!} : \mathit{YES} \mid \mathit{NO} \\
\hline
\mathit{whichnode?} \mapsto b \in \mathit{nodeGoals} \Rightarrow \forall\, c \in b \mid c \mapsto \mathit{BUSY} \notin \mathit{states} \\
\mathit{whichnode?} \mapsto d \in \mathit{nodeFiles} \Rightarrow \forall\, e \in d \mid e \mapsto \mathit{BUSY} \notin \mathit{states} \\
\mathit{result!} = \mathit{NO} \\
\hline
\end{array}$$

$$\mathit{Is\_Busy} \mathrel{\widehat{=}} \mathit{Busy\_Yes} \;\vee\; \mathit{Busy\_No}$$

Upon all of a node's children leaving the *BUSY* state, a node must determine if it has itself changed, and then set its state accordingly. The schema which represents this operation is defined next.

$$
\begin{array}{l}
\textit{End\_Spawn} \\
\hline
\Delta \mathit{Manager} \\
\mathit{anode}? : \mathit{NODE} \\
\hline
\mathit{anode}? \mapsto \mathit{CHANGED} \in \mathit{states}' \Rightarrow (\\
\quad \mathit{anode}? \mapsto a \in \mathit{nodeGoals} \Rightarrow \exists\, b \in a \mid b \mapsto \mathit{CHANGED} \in \mathit{states}' \land \\
\quad \mathit{anode}? \mapsto c \in \mathit{nodeFiles} \Rightarrow \exists\, d \in c \mid d \mapsto \mathit{CHANGED} \in \mathit{states}' \land \\
\quad \mathit{anode}? \mapsto a \in \mathit{nodeGoals} \Rightarrow a = \varnothing \land \\
\quad \mathit{anode}? \in \mathit{files} \Rightarrow (\mathit{anode}? \mapsto e \in \mathit{datestamps} \ \land \ \mathit{anode}? \mapsto e \notin \mathit{datestamps}')) \\
\mathit{anode}? \mapsto \mathit{NOT\_CHANGED} \in \mathit{states}' \Rightarrow (\\
\quad \mathit{anode}? \mapsto r \in \mathit{nodeGoals} \Rightarrow \forall\, s \in r \mid s \mapsto \mathit{NOT\_CHANGED} \in \mathit{states}' \land \\
\quad \mathit{anode}? \mapsto t \in \mathit{nodeFiles} \Rightarrow \forall\, u \in t \mid u \mapsto \mathit{NOT\_CHANGED} \in \mathit{states}' \land \\
\quad \mathit{anode}? \in \mathit{files} \Rightarrow (\mathit{anode}? \mapsto v \in \mathit{datestamps} \ \land \ \mathit{anode}? \mapsto v \in \mathit{datestamps}')) \\
\hline
\end{array}
$$

Only goals can have commands associated with them. Therefore, upon successfully determining that a goal has changed it must execute the commands associated with it. Files, on the other hand, must be copied locally so that they can be included in the compile if they have been changed or need to be updated. The copyFile(file) function appears in the following schema which will perform this copy to the local host.

$$
\begin{array}{l}
\textit{Execute\_Commands} \\
\hline
\Xi \mathit{Manager} \\
\mathit{whichnode}? : \mathit{NODE} \\
\hline
\mathit{whichnode}? \mapsto \mathit{CHANGED} \in \mathit{states} \Rightarrow (\\
\quad \mathit{whichnode}? \mapsto x \in \mathit{commands} \ \land \ x \neq \varnothing \Rightarrow \mathit{execCommands}(x) \lor \\
\quad \mathit{whichnode}? \in \mathit{files} \Rightarrow \mathit{copyFile}(\mathit{whichNode}?)) \\
\hline
\end{array}
$$

Finally after execution is finished, the state of the nodes is reset to empty. This is so that there are no states allocated during the majority of the time where a system is inactive or a user is viewing or changing dependencies. The schema which expresses this behavior follows.

$$
\begin{array}{l}
\textit{End\_Execute} \\
\hline
\Delta \mathit{Manager} \\
\hline
\mathit{states}' = \varnothing \\
\hline
\end{array}
$$

4.4.7 Saving and Loading

Some of the most essential operations in the system are the save and load operations. These allow a user to establish their dependency tree and save it for future use, such as upon next execution of this product. This also allows for the data file to be distributed and other users to use the same data file that are working on the same project.

Outlined previously were all the operations of adding and querying of the system. These operations provide a clear and concise way of checking the data structures described in the state schema for the Manager.

A save operation merely takes all of the data elements provided in the Manager state schema and dumps them into the format of the data file, shown in Section 4.5. The query operations defined previously allow for easy access to these data elements and for this reason it is felt that the specification of this operation is both redundant and tedious.

The load operation is similar. The given data file in the standard format must be searched and the data elements of the Manager must be established. The operations were clearly defined above to allow for easy addition of the data to the system. The task consists of parsing the data file and determining where the elements of the data file go in the Manager and then placing them there. For this reason, it is again felt that the specification of this operation is redundant and unnecessary.

The level of abstraction for this system is established as above this. With the clearly defined operations to add and query this system, there is no need to specify the simple options of saving and loading the data file.

4.4.8 Converting

Conversion of a makefile into the native format is quite difficult. In implementation there are many aspects an *ADVANCED* makefile which are beyond the scope of this project. Makefiles can be created in a manner which resembles programming and the parsing and symantic evaluation of those files would require so much time that it would detract from this project's true goal. Therefore, this product will only allow for basic makefiles to be converted. While this might seem like a large restriction on the system, it is important to note that the vast majority of *Make* users use this product to simply define a dependency tree (not as advanced as that allowed by this product) and to compile their product. Some even defined argument which can be invoked from the command line. This product handles those arguments by making them *Top Level Goals* with the given commands. These are then given (as with all goals) to the user so that they can be executed upon command.

The operation to convert an arbitrary makefile into the native format is therefore an extremely difficult one to implement. The operations to take the data from the makefile are the ones that are not established. Once the data has been retrieved from the makefile, it can be placed into the native format simply by using the well defined operations defined in the specification such as *AddTopLevelGoal*, *AddNodeGoal*, *AddNodesCommands*, and others.

There must be a way to get all the top level goals of a makefile. Once these have been retrieved, the *AddTopLevelGoal* operation defined previously can be used. Then, there must be operations to get all the commands of a goal, all the other goals of a goal, and finally all the files of a goal. Since *Make* does not allow for files to be located across the network, locations of the files are irrelevant, as are logins to URL sites.

The state of a makefile can be established now. This uses the set $[NODE]$ defined in the Manager specification which is again the set of all nodes in existence. The set of commands $[COMMAND]$ defined in the Manager specification is also used here. The Makefile specification has all the top level goals defined in the makefile, the files associated with a node, the goals associated with a node, and the commands associated with a node. The strict control over whether a file has commands is not controlled here as the makefile format controls this. Should a user enter invalid data into a makefile, then the schemas which control the adding of this data to the Manager system will fail.

Makefile

$$\begin{array}{l}
nodes : \mathbb{P}\, NODE \\
goals : \mathbb{P}\, NODE \\
files : \mathbb{P}\, NODE \\
toplevelgoals : \mathbb{P}\, NODE \\
nodeFiles : \mathbb{P}(NODE \pfun \mathbb{P}\, NODE) \\
nodeGoals : \mathbb{P}(NODE \pfun \mathbb{P}\, NODE) \\
commands : \mathbb{P}(NODE \pfun \mathbb{P}\, COMMAND) \\
\hline
nodes = toplevelgoals \cup goals \cup files \\
\forall\, g \in goals \Rightarrow g \notin files \land g \notin toplevelgoals \\
\forall\, f \in files \Rightarrow f \notin goals \land f \notin toplevelgoals \\
\forall\, a \in toplevelgoals \Rightarrow a \notin goals \land a \notin files
\end{array}$$

The initial state of the Makefile can now be established for consistency.

Makefile_Init

$$\begin{array}{l}
Makefile' \\
\hline
nodes' = \varnothing \\
goals' = \varnothing \\
files' = \varnothing \\
toplevelgoals' = \varnothing \\
nodeFiles' = \varnothing \\
nodeGoals' = \varnothing
\end{array}$$

A proof of the initial state of the *Makefile* schema is provided next. Here it is shown that the data invariants are not violated.

(1)	$nodes' = \varnothing$	(premise)
(2)	$goals' = \varnothing$	(premise)
(3)	$files' = \varnothing$	(premise)
(4)	$toplevelgoals' = \varnothing$	(premise)
(5)	$nodeFiles' = \varnothing$	(premise)
(6)	$nodeGoals' = \varnothing$	(premise)
(7)	$\varnothing = \varnothing \cup \varnothing \cup \varnothing$	((1), ident.)
(8)	$\varnothing = toplevelgoals \cup goals \cup files$	((2), (3), (4), subst.)
(9)	$nodes = toplevelgoals \cup goals \cup files$	((1), subst.)
(10)	$(\forall g \in \varnothing \Rightarrow g \notin \varnothing \land g \notin \varnothing$	(ident.)
(11)	$(\forall g \in goals \Rightarrow g \notin files \land g \notin toplevelgoals$	((2), (3), (4), subst.)
(12)	$(\forall f \in \varnothing \Rightarrow f \notin \varnothing \land f \notin \varnothing$	(ident.)
(13)	$(\forall f \in files \Rightarrow f \notin goals \land f \notin toplevelgoals$	((2), (3), (4), subst.)
(14)	$(\forall a \in \varnothing \Rightarrow a \notin \varnothing \land a \notin \varnothing$	(ident.)
(15)	$(\forall a \in toplevelgoals \Rightarrow a \notin goals \land a \notin files$	((2), (3), (4), subst.)

Q.E.D.

The operation to get all the top-level goals of the makefile can now be defined. It passes as output all the top level goals defined in the makefile. These are usually the default (no argument) target and the command line argument targets defined in the makefile.

Makefile_GetTopLevelGoals

$\Xi Makefile$
$out! : \mathbb{P}\ NODE$

$out! = toplevelgoals$

Here the same response type used in the Manager specification is defined again.

$RESPONSE ::= SUCCESS \mid FAILURE$

In *Make*, goals can have dependencies of other goals. Because of this, there must be an operation to get the dependency goals of a specified node. This is defined next.

Makefile_GetNodesGoals_OK

$\Xi Makefile$
$aNode? : NODE$
$out! : \mathbb{P}\ NODE$
$res! : RESPONSE$

$aNode? \in goals \lor aNode? \in toplevelgoals$
$aNode? \mapsto b \in nodeGoals$
$out! = b$
$res! = SUCCESS$

This operation will fail if the given node is not either a top level goal or a goal defined in the makefile. The schema that describes this behavior follows.

$Makefile\_GetNodesGoals\_ERROR$

$\Xi Makefile$
$aNode? : NODE$
$res! : RESPONSE$

$aNode? \notin goals \wedge aNode? \notin toplevelgoals$
$res! = FAILURE$

$$Makefile\_GetNodesGoals \mathrel{\widehat{=}} Makefile\_GetNodesGoals\_OK \wedge Makefile\_GetNodesGoals\_ERROR$$

Similarly, goals can have dependencies of files. Top level goals cannot have file dependencies, however, just regular goals. Therefore, there must be an operation which retrieves the file dependencies of a particular goal. This operation is defined next.

$Makefile\_GetNodesFiles\_OK$

$\Xi Makefile$
$aNode? : NODE$
$out! : \mathbb{P}\ NODE$
$res! : RESPONSE$

$aNode? \in goals$
$aNode? \mapsto b \in nodeFiles$
$out! = b$
$res! = SUCCESS$

This operation will fail if the given node is not a regular goal defined in the makefile. The schema that describes this operation follows.

$Makefile\_GetNodesFiles\_ERROR$

$\Xi Makefile$
$aNode? : NODE$
$res! : RESPONSE$

$aNode? \notin goals$
$res! = FAILURE$

$$Makefile\_GetNodesFiles \mathrel{\widehat{=}} Makefile\_GetNodesFiles\_OK \wedge Makefile\_GetNodesFiles\_ERROR$$

Goals have associated with them commands, which are executed in the makefile. These can be associated with either top level goals, or regular goals. Next is the operation which defines how to get the commands associated with a given node.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Makefile\_GetGoalsCommands\_OK}} \\
\hline
\Xi \mathit{Makefile} \\
aGoal? : NODE \\
out! : \mathbb{P}\ COMMAND \\
res! : RESPONSE \\
\hline
aGoal? \in goals \lor aGoal? \in toplevelgoals \\
aGoal? \mapsto b \in commands \\
out! = b \\
res! = SUCCESS \\
\hline
\end{array}$$

This operation will fail if the given goal is not in the system. Since, in *Make*, all goals have associated with them a command, there is no need to have an error where the goal has no commands. Next is the operation which defines the forementioned error.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Makefile\_GetGoalsCommands\_ERROR}} \\
\hline
\Xi \mathit{Makefile} \\
aGoal? : NODE \\
res! : RESPONSE \\
\hline
aGoal? \notin goals \land aGoal? \notin toplevelgoals \\
res! = FAILURE \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{Makefile\_GetGoalsCommands} \mathrel{\widehat=} \mathit{Makefile\_GetGoalsCommands\_OK} \land \\
\mathit{Makefile\_GetGoalsCommands\_ERROR}
\end{array}$$

4.5 The Datafile

In order to store all of the necessary dependencies and locations of files for a total system, a standard datafile has been established. This datafile is a textual listing of all of the goals, files, and options for a system and is saved as a user-defined title with a *.dat extension.

Each of the sections of the datafile will be delineated by the keywords *OPTIONS*, *GOALS*, and *FILES*. These keywords will mark the start of each section, as shown below in the example datafile for the EDIT system introduced in Section 3.1.

```
# Group One   June 3, 1998
# Sample datafile

#---------------------------------------------------------------
OPTIONS

OBJECT_FILE: "/local/project/object"
```

```
#-------------------------------------------------------------
GOALS

FILENAME: "edit"
FILES:
GOALS: {"main", "kbd", "command", "display",
        "insert", "search", "files", "utils"}
COMMANDS: {"cc -o edit main kbd command display
        insert search files utils"}

FILENAME: "main"
FILES: {"main.c"}
GOALS:
COMMANDS: {"cc -c main.c"}

FILENAME: "kbd"
FILES: {"main.c"}
GOALS:
COMMANDS: {"cc -c kbd.c"}

FILENAME: "command"
FILES: {"command.c"}
GOALS:
COMMANDS: {"cc -c command.c"}

FILENAME: "display"
FILES: {"display.c"}
GOALS:
COMMANDS: {"cc -c display.c"}

FILENAME: "insert"
FILES: {"insert.c"}
GOALS:
COMMANDS: {"cc -c insert.c"}

FILENAME: "search"
FILES: {"search.c"}
GOALS:
COMMANDS: {"cc -c search.c"}

FILENAME: "files"
FILES: {"files.c"}
GOALS:
COMMANDS: {"cc -c files.c"}
```

```
FILENAME: "utils"
FILES: {"utils.c"}
GOALS:
COMMANDS: {"cc -c utils.c"}

FILENAME: "clean"
FILES:
GOALS:
COMMANDS: {"rm edit main kbd command display insert search files utils"}

#-----------------------------------------------------------------
FILES

FILENAME: "main.c"
FILES: {"defs.h"}
GOALS:
LOCATION: "file:/usr/main.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "kbd.c"
FILES: {"command.h", "defs.h"}
GOALS:
LOCATION: "<userid>:<passwd>@ftp://usr/project/kbd.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "command.c"
FILES: {"command.h", "defs.h"}
GOALS:
LOCATION: "<userid>:<passwd>@ftp://usr/project/command.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "display.c"
FILES: {"defs.h"}
GOALS:
LOCATION:  "http://www.ufl.edu/~usr/project/display.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "insert.c"
FILES: {"buffer.h", "defs.h"}
GOALS:
LOCATION: "http://www.ufl.edu/~usr/project/insert.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "search.c"
FILES: {"buffer.h", "defs.h"}
GOALS:
```

```
LOCATION: "http://www.yahoo.com/search.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "files.c"
FILES: {"command.h", "defs.h", "buffer.h"}
GOALS:
LOCATION: "file:/usr/project/files.c"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "utils.c"
FILES: {"defs.h"}
GOALS:
LOCATION: "http://www.ufl.edu/~usr/project/utils.o"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "command.h"
FILES:
GOALS:
LOCATION: "<userid>:<passwd>@ftp://usr/project/command.h"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "defs.h"
FILES:
GOALS:
LOCATION: "http://www.company.com/project/defs.h"
MOD_DATE: "09 Jun 1998 21:46"

FILENAME: "buffer.h"
FILES:
GOALS:
LOCATION: "file:/usr/project/buffer.h"
MOD_DATE: "09 Jun 1998 21:46"
```

In this datafile, the '#' represents a commented line. The programmers can determine the actual symbol(s) used to mark comments, as long as a standard is determined early in the development stage and remains consistent throughout the system. Again, this datafile is meant to serve as a mere prototype for a final version. It is difficult to ascertain at this point the additions or alterations that may become necessary depending on the implementation. Quotes may need to be added or removed from various portions of the datafile in order to comply with the semantics of string inputs in the chosen implementation language. All of this is acceptable and expected.

Within each section of the datafile, each node is represented as a listing of attributes associated with that node. Strings listed in all caps followed by a colon represent the attribute names. The value associated with each attribute follows the colon and is, in this case, surrounded by quotes if the value is a string, and surrounded by brackets if the value is a list. A blank line delineates a single node from a consecutive node.

The *OPTIONS* section is a means by which the software designers may chose to include any number of options that the user may select for the given dependency tree. The above example lists an attribute entitled OBJECT_FILE, which is followed by a location on the local server. This option is intended to represent the user's desire to store all of the object files of the current design in one given location. This option should always be activated, however a default location of pwd/object should always be in place unless the user designates otherwise.

Within the *GOALS* section, it can be noted that neither the location, nor the modification date attributes are associated with any goal. This is because these attributes are invalid within the context of a goal. All goals are stored on the local server due to the various architectural restrictions that many systems impose on a goal. Furthermore, the new datestamp system that has been developed for this system deals solely with the datestamp associated with a given FILE (see Section 4.6). Therefore, any datestamp that could be associated with a goal is unnecessary, and would only lead to confusion within the system, or with the users themselves. For these reasons, the LOCATION and MOD_DATE attributes have been omitted from the *GOALS* section of the datafile.

The *FILES* section of the datafile contains a similar omission of an attribute. The example reveals an omission of the COMMANDS attribute. The definition of a file clearly states that a file is a freestanding source file. Simply put, a source file cannot be generated using any form of a command or set of commands, as its entire function is a means for some higher goal to be met. The definition itself clearly depicts the reasoning behind this attribute exclusion.

An important property of the datafile as a whole is that while it is typically handled by the Configuration Management system, it is also completely editable by an outside user. Rather than launching the entire system and making adjustments to a dependency tree via the graphical user interface, a user has the option of making adjustments to the datafile directly. Of course, this functionality opens the doors for numerous user created errors. The users manual will recommend that only advanced users attempt to manually alter the datafile. Furthermore, upon loading a given datafile, if it is determined that an invalid field is contained in the datafile, the user will be notified of a corruption. This error dialog will then specify the line on which the error occurred and provide a brief description of the error, similar to a typical compiler.

4.6 Datestamps

Due to the fact that the Configuration Management system should provide for files to be located at any accessible location in the world, a revolutionary new system has been developed to determine whether or not a given file requires recompilation. This new system, referred to as the Datestamp system, works in conjunction with the datafile to produce sound methodology for overcoming the dreaded "synchronization problem."

Before elaborating on this new datestamp concept, it may be helpful to digress and explain how the archaic *make* system has handled the recompilation decisions in the past.

The old *make* system would perform a series of comparisons of the time attached to a node and each of its children. If any of a node's children turned out to be newer than the node itself, i.e. the child had been compiled after the node, the node would recompile. Of course, this was a recursive operation, where a given child would compare its own timestamp to that of its children to determine its own need for recompilation before allowing its parent to recompile. This system worked only because all of the files were stored on a single system. This method clearly is not appropriate for the new network accessible system. Various problems would arise if network clocks were not synchronized, or if systems were located in separate time zones.

The new Datestamp system is based on a completely different foundation. The need to recompile is determined by querying the file for it's last recorded modification time. This modification time is compared to the expected modification time for that file that was stored in the datafile during the last query. If the two times are not completely identical, this will signal the system that the file has been updated, and all goals that depend on this file should be recompiled.

As the system is initialized, all of the datestamps are set to NEW_DATE, which is time zero (Jan.1 , 1970) in UNIX systems. Clearly, upon first pass, all of the nodes of a design will appear out-of-date to the Configuration Management system, because none of the last revision datestamps of the files will match their NEW_DATE stored in the datafile. Therefore, all files will recompile upon first pass into the system. This initial recompilation is not only acceptable, but essential, because upon an initial pass none of the remote source files would have been brought to the local server for the necessary local compilation. In short, none of the remote files would have local object files, so the system would not work without these necessary object files anyway.

If, however, the user does not wish to perform this initial compilation pass for whatever reason, they can manually enter the most recent revision time into the datafile. In this case, the system datestamp associated with a file and its expected datestamp located in the datafile will match, wherefore recompilation will not occur. Similarly, if a user wishes for the system to recompile despite the fact that a file has not been updated, a manual alteration of a datestamp within the datafile to a value other than the actual last revision time will always cause recompilation.

If a case arises that a remote system is not POSIX compliant and thus cannot return or associate a last modified date with a file, a datestamp of null will be returned. The null datestamp will never match the initial NEW_DATE associated with every file in a new system, therefore the CM system will always bring the file to the local host and recompile.

This new Datestamp system carries many benefits. The first benefit is that it alleviates the problem of files being stored in differing time zones. It is quite possible that many engineers spread across the country or even across the world could be working on a single project. While it is true that many systems adjust for time zone differences, it should not be assumed that this is always the case.

Furthermore, even if all systems did adjust for time zone differences, this would in no way eliminate the "synchronization problem." It is not realistic to assume that all servers have perfectly synchronized clocks. In fact many systems may differ by several seconds or even several minutes. Many systems might even be flat out wrong as to the current date and time. Because the only time comparison performed would be relative to the last time recorded by the same remote system, synchronization is clearly not necessary.

Finally, the old make system suffers from a major flaw. An adjustment of the system clock will produce invalid compile times. These compile times may result in an incorrect decision not to recompile, when in fact recompilation is necessary. In the new system, it does not matter if a revision time is newer or older than any of the other files, it just matters if it is not the expected time for that given file itself. This methodology is far superior in that it does not suffer from the possible problems associated with system clock adjustment.

4.7 Compilation

4.7.1 Compilation Order Rules

One of the requirements of the Manager was that UNIX *Make* and Ada compilation order rules be observed. The orders of compilation for both systems are the same, only the methods in which dependencies are handled are different. The compiler handles dependencies for Ada, while *Make* checks for dependencies before compilation begins. Since the Manager is upwardly compatible with *Make*, it also checks dependencies before execution. Software configurations in which these rules apply can be arranged in a DAG, where each node depends on its children. In order for any node to be compiled, its children must first be updated so any changes will be reflected in the new compilation. A child is updated if its modification date has changed, or if any of its children have been updated. Ada's compiler checks for this, as well as *Make* and the Manager. A feature the Manager has that the others do not have is the ability to maintain a configuration across a network. It can access a file through a URL and copy it if the date has changed.

4.7.2 Compilation Procedure

In the Manager, the algorithm for compilation begins with the user choosing a node in the DAG from which to begin compilation. This node becomes the compilation root as mentioned in Section 4.4.6. A new thread is spawned for each of the root's children. Since a child can have more than one parent, the possibility of redundant threads for that child being spawned must be managed. Therefore a semaphore is used to let the parent know the status of the child before spawning a new thread. The possible states a semaphore can have are

BUSY, *CHANGED*, *NOT_CHANGED*, and *NOT_HIT*. The parent node will check the state of its child and spawn a thread if the state is *NOT_HIT*. This check/spawn operation must be atomic to prevent another parent from spawning a thread between the time the state is checked and the thread is spawned. The addition of this semaphore makes it necessary to set the state of all the semaphores to *NOT_HIT* before compilation begins to allow all of the nodes to be processed. After the states have been set, the parent (with the top-level goal as the first parent) sets its state to busy and spawns a thread for all of its children whose state is *NOT_HIT*. The parent then waits for all of the children to end their *BUSY* state. If any of the parents' children's state is *CHANGED*, or if the parent is a file and the file's date has changed, the parent sets its state to *CHANGED*. Barring these conditions, the parent will set its state to *NOT_CHANGED*. After the parent has changed its state, it executes its commands or copies the file if the state has been changed to *CHANGED*. The coping of the file is done so that it will be on the local machine when compiled or needed to be included in a compile, such as would be the case with a header file. Compilation ends when the compilation root (in most cases the root of the tree) notices that all of its children have either changed or not changed. At this point, it executes the commands associated with itself. Finally, all the states of the nodes are set to NULL so as not to keep the state variable allocated when it is not needed.

The spawnThread() operation will fork with the parent returning to wait on it's children, and the child calling a compile() on itself. This operation is recursive-like in that each parent spawns all of their children and then must wait for the children to return before it can set its own state. The leaves will spawn no threads and set their state immediately. The new state will be determined the same as any other node in the DAG. Once the leaves have set their state, the parents will set their state until the top-level goal has set its state and finishes the compile.

The following pseudocode describes this process in coded form, for increased understanding:

```
void initCompile(Node rootNode)
{
   setAllNodeStates(NOT_HIT);
   compile(rootNode);
   endCompile();
}

void compile(Node currentNode)
{
   initSpawn(currentNode);
   isBusy(currentNode);
   endSpawn(currentNode);
}
```

```
void initSpawn(Node currentNode)
{
   currentNode.setState(BUSY);
   for i=0 to currentNode.numGoals()
      if (currentNode.getGoal(i).currentState() == NOT_HIT)
         spawnThread(currentNode.getGoal(i));
   for j=0 to currentNode.numFiles()
      if (currentNode.getFile(j).currentState() == NOT_HIT)
         spawnThread(currentNode.getFile(j));
}

void isBusy(Node currentNode)
{
   BOOL stillBusy = TRUE;

   while(stillBusy)
   {
      int  numberBusy;
      numberBusy =0;

      for i=0 to currentNode.numGoals()
         if (currentNode.getGoal(i).currentState() == BUSY)
            numberBusy++;
      for j=0 to currentNode.numFiles()
         if (currentNode.getFile(j).currentState() == BUSY)
            numberBusy++;

      if (numberBusy == 0)
         stillBusy = FALSE;
   }
}

void endSpawn(Node currentNode)
{
   Bool childChanged = FALSE;

   for i=0 to currentNode.numGoals()
      if (currentNode.getGoal(i).currentState() == CHANGED)
         childChanged = TRUE;
   for j=0 to currentNode.numFiles()
      if (currentNode.getFile(j).currentState() == CHANGED)
         childChanged = TRUE;

   if (currentNode.isFile() && currentNode.dateChanged())
      currentNode.setState(CHANGED);
   if (childChanged == TRUE)
```

```
      currentNode.setState(CHANGED);
   else if (childChanged == FALSE)
      currentNode.setState(NOT_CHANGED);

   execCommands(currentNode);
}

void execCommands(Node currentNode)
{
   if (currentNode.currentState() == CHANGED)
      if (currentNode.isGoal() && currentNode.getCommands() != NULL)
         System(currentNode.getCommands());
      else if (currentNode.isFile())
         copyFile(currentNode.getLocation());
}

void endCompile();
{
   setAllNodeStates(NULL);
}
```

4.8 Graphical User Interface

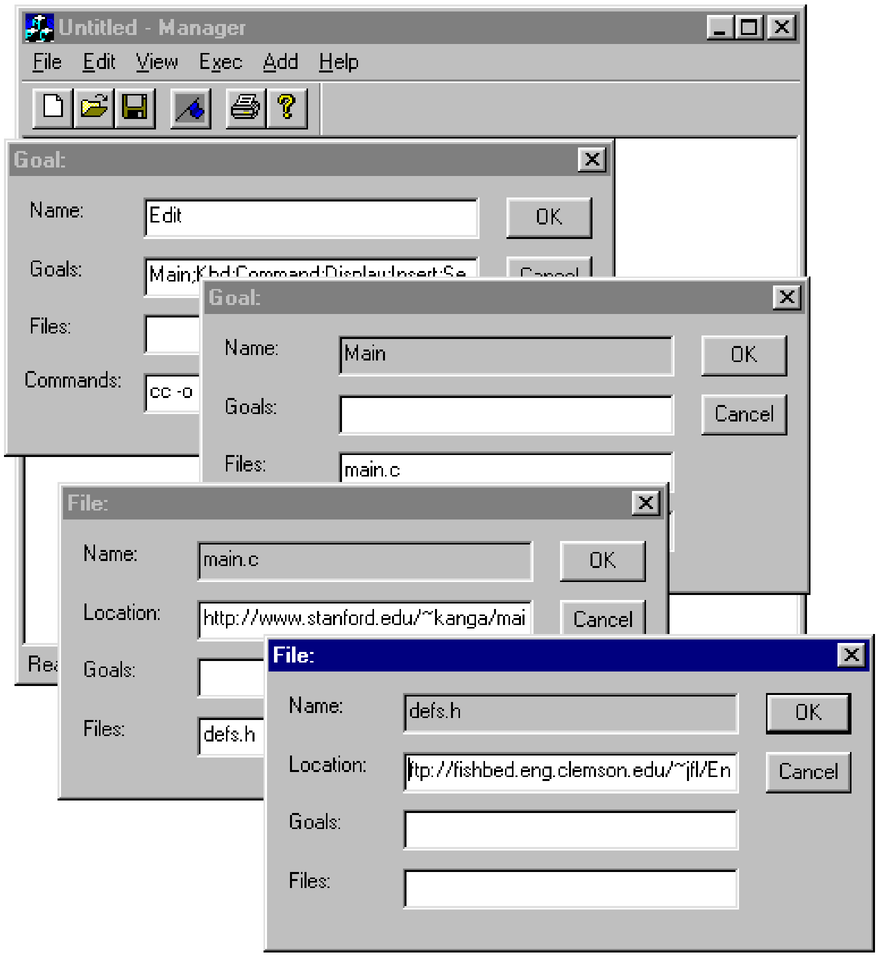
The Graphical User Interface is what allows the user to interact with the system. The GUI should be user-friendly, and accommodate the novice as well as expert user. The Manager's specifications are such that the GUI can be implemented in any language such as Java, or Visual C++. The interface will enforce data protection with methods for the user to add, delete, or edit nodes. This prevents the user from attempting to modify something that can not exist, such as a goal's location. It also prevents the user from modifying system variables such as a file's last modification date. The interface should be intuitive and provide differing ways to invoke a method such as a toolbar and drop down menu. The GUI will also provide a graphical representation of the dependency DAG as seen in Figure 3.2. This DAG should be interactive by allowing the user to edit, view, or compile a node in the DAG by clicking it with the mouse. Other options or attributes of the GUI include dialog boxes for adding goals and nodes (Figure 4.1), a hierarchy view of dependencies in the drop down menu (Figure 4.2), and a hierarchy view of goals for compilation in the drop down menu (Figure 4.3).

The dialog box should add the node to the DAG, then immediately prompt the user for information on that node's dependencies (as seen in Figure 4.1). In this example, the sample makefile from Section 3.1 was used. The user adds a goal by clicking the flag button or selecting add goal from the drop down menu. The user adds Edit as the root goal and supplies any dependencies and commands for that goal. When OK is pressed, the list of goals and files that depend on Edit is parsed and a new dialog pops up for information on each new node. In this case Main is the first goal in the list. Information for Main is entered and a dialog for main.c pops up because Main depends on main.c. Information for main.c is entered and a dialog for defs.h pops up. After the information for defs.h is entered, a new dialog will pop up for Kbd, because Main and main.c only have one child and defs.h has none. This will continue until information for all of Edit's children or dependencies has been entered. Essentially, the functionality of the dialog boxes is that fields are entered in a depth-first manner.

The hierarchy view of the dependencies (Figure 4.2) allows the user to get a visual idea of the hierarchy and could allow the user to edit any given node. Another means for representing the dependency tree is the graphical representation presented in Figure 3.2 and Figure 3.4.

The hierarchy view of the goals in Figure 4.3 allows the user to begin compilation from any arbitrary goal.

The GUI must implement all of the interfaces the user requires, while restricting access to those the user does not. It must also provide adequate help for the user as well as notes on default assumptions (such as the initial date, as explained in Section 4.4.4) and limitations. Above all, it must conform to the Manager's specification.



Untitled - Manager

Goal:
Name: Edit
Goals: Main;Kbd;Command;Display;Insert;Se
Files:
Commands: cc -o

Goal:
Name: Main
Goals:
Files: main.c

File:
Name: main.c
Location: http://www.stanford.edu/~kanga/mai
Goals:
Files: defs.h

File:
Name: defs.h
Location: ftp://fishbed.eng.clemson.edu/~jfl/En
Goals:
Files:

Figure 4.1: Adding Goals and Files to the System

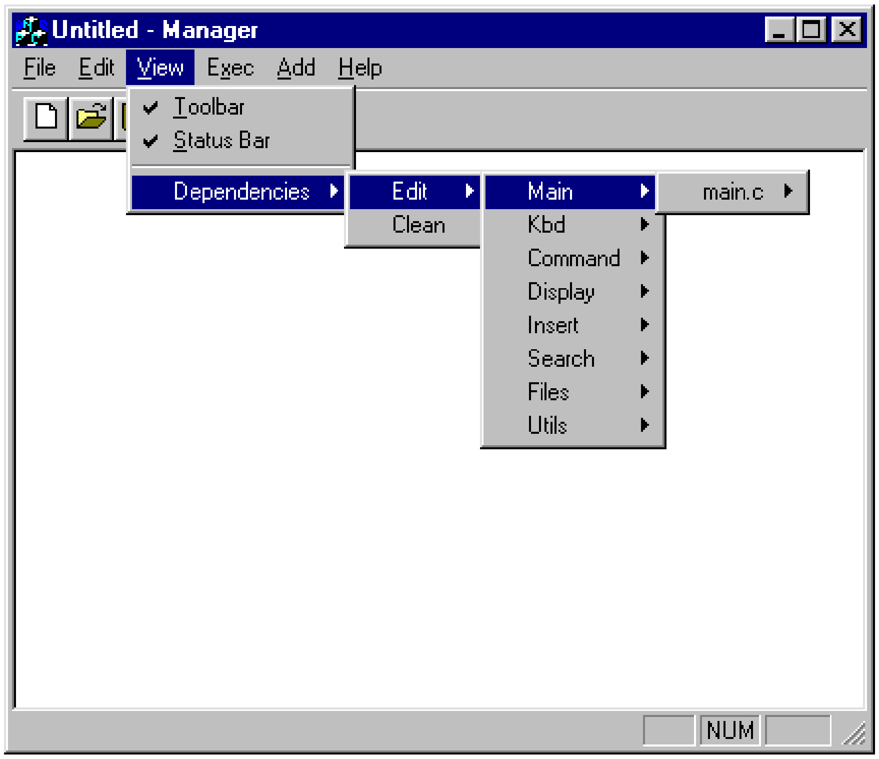


Figure 4.2: Viewing Goals and Files of the System

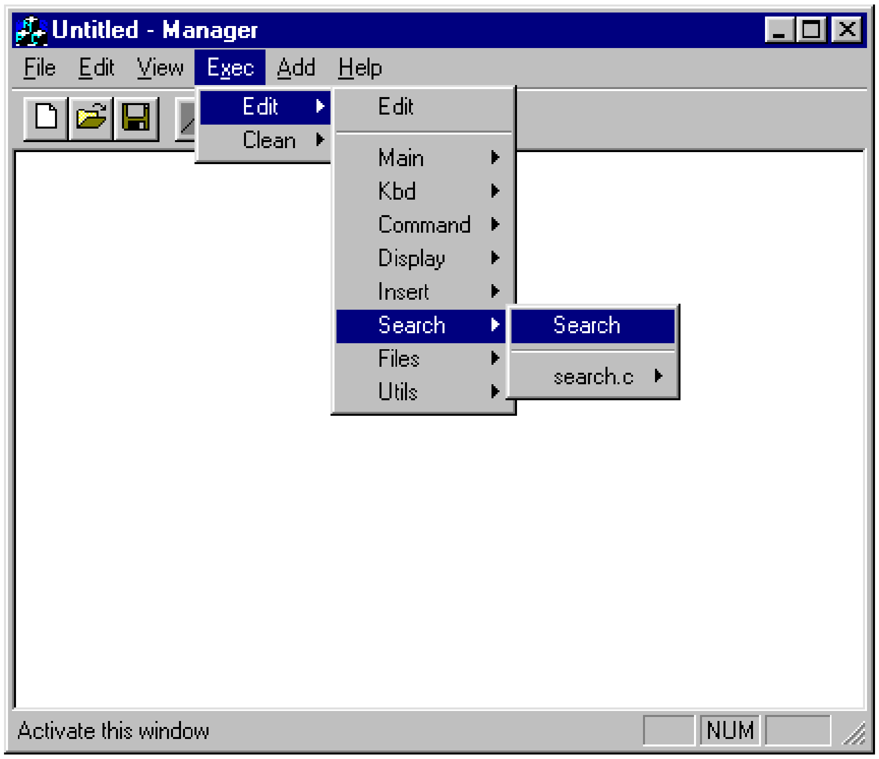


Figure 4.3: Compiling Goals in the System

Chapter 5

Project Properties

There were a number of goals which were to be achieved during the course of this project. As with any project where a set of predefined requirements is to be met, these requirements must be shown to have been satisfied by the implementation. What follows are the requirements and discussion on how they have been met.

5.1 Statement of Need

These days, more and more companies have made the move toward global networking. Because of the speed and ease at which a company can communicate with other parts of the country, or the world at large, it is no longer necessary to keep even the engineers for a single project in one central location. Companies are now able to capitalize on the economic benefits of having several plants spread across the world.

This trend has brought about a need for the Next Generation Software Configuration Management systems. These systems should not only provide the functionality of the standard *Make* system, but also incorporate a large degree of network accessibility.

The Configuration Management System presented in this paper meets the two essential goals - *Make* upward compatibility and complete network accessibility. The new system includes corrections to most, if not all, of the previously non-network-compliant attributes of *Make* while also providing a user-friendly graphical interface. For the expert users of the product who feel speed is more valuable than user-friendliness, the faster text-based editing procedures do the job particularly well.

Clearly, the completed version of the system specified in this paper will soon become an essential piece of software for any large software project. Various companies have expressed an interest in a piece of software with the features employed in this system.

5.2 Expected Market

While the initial market for this Next Generation Software Configuration Management system may be limited to large companies with projects spanning the globe, it should soon become a standard piece of software, completely replacing *Make* and other similar systems. The new method for dealing with datestamps exhibits a functionality that could be beneficial to any given project, large or small, local or spread across the network. This and many other features included within the new Software Configuration Management System will quickly distinguish it as a powerful and useful tool throughout the market.

5.3 Formal Methods

This entire paper takes on the form of a large formal specification. Over 30 Zed schemas are included in the document, which clearly define any and all operations that will be incorporated into the final system. Some of the vital operations are verified using formal proofs. Theoretically, the entire Configuration Management system described in this document could be created completely and correctly using the specification and text included in the document alone.

5.4 Test Specifications

Numerous prototypes are included within the text of this document. These prototypes, together with the abundant sample graphics should clarify the expected behavior of the final system to the novice user. Furthermore, the final product calls for a thorough tutorial to familiarize a user with the expected behavior of the system. A user's manual is also called for, which should demonstrate the complete functionality of the system, while serving as a handy reference tool.

These are not presented here, as this design project was not implementation of the product but of design. Therefore, instructions on how to install the product are impossible. However, throughout the paper the expected control and functionality to the end user have been expressed.

5.5 Self-Realization

The final system, once created, will have the ability to manage itself, just as it would manage any arbitrary system of files. That is, it will be self-realizable.

Self-realization is possibly the ultimate test of a product. Many products cannot be self-realizable simply from the nature of the product. For instance, while a type setting or word processing program can be used to write the documentation for itself, an operating system cannot be used in a self-realizable way. This project lends itself uniquely to being extremely self-realizable. Once the product has been implemented, then it clearly will contain many source files, perhaps some header files, and design documentation. Upon successfully creating an executable, the product could then be used to manage itself by using the interfaces defined in this document to load and manage the source, header, and other files used by this project. In this way, the product would be self-realizable. Lack of implementation limits the proof of this property, however.

5.6 Portability

Varying levels of portability can be achieved depending on the language used to implement this product. Portability levels could range from the restrictive nature of an Ada program, to the limitless portability of a Java program. At this point in time, it would be recommended that Java be used to create this system, due to its compliance with Sun's 'Write Once, Run Anywhere' goal. However, the specification is flexible enough to allow for the system to be produced by any high-level programming language that may arise in the near future.

All operations defined within the text of this document are based on the POSIX interface. In this respect, the Configuration Management system is completely portable. Furthermore, this system facilitates the use of any arbitrary compiler, without any more effort than it would take to type the compile command at a shell prompt.

5.7 Software Reuse

The Configuration Management system builds on numerous systems introduced in the past. The similarities to the *Make* system, and the new system's ability to work properly given a standard makefile are but small examples. Limitless instances of software reuse can, and should, be incorporated into the final product. The Java Developers Toolkit includes a URL class, which could be utilized to easily access source files on remote systems as defined in the Network Specification, Section 4.2. An example of an implementation of the URL class is included in Appendix B. Furthermore, a means for creating the graphical node representation as specified in the Example Makefile Section, Section 3, is currently being created in the Parallel Architecture lab at Clemson University and could be reused in the creation of this final product.

5.8 Efficiency

Although the time it takes to query and retrieve a remote file may seem lengthy, this drawback can be counteracted by the incorporation of vast amounts of parallelism that can be achieved within the typical compile operation. The fact that the new system incorporates the use of threads during it's execution, and will compile various independent files simultaneously, will result in an exponential performance increase.

Using the breadth-first search described in Section 4.4.6 and Section 4.7, all of a node's children are spawned at the same time. This allows for each thread to be run in parallel. Then, should a particular node be slowed down by the network for any reason, the system continues to operate and performance is not sacrificed.

5.9 Generality

As described in Section 4.3, this system does not employ version control. While this was not required in the design, it is becoming an important part of many software design projects. Every attempt has been made to allow for this product to be upgradable to employ revision control if it is necessary in later versions.

Appendix A

Specific Requirements Paper

A.1 Overview

A software configuration management system will be designed. This software package will be one that eases the job of compiling and maintaining large software projects. Network support, *Make* compatibility and a user friendly interface are just a few of the properties of this system. A formal specification language will be employed to show consistency and completeness.

The advantage of this system is that it allows for users to specify a dependency tree between files. These files can be source code, header files, or even design documents. Regardless, the order of compilation and dependencies will be maintained through using the date a file was last modified.

A.2 Features

The configuration management system has many features but most can be broken into two sections: local and network features. The local features are those which are the core of the system while the network features allow for the retrieval and compilation of files across the network.

A.2.1 Local

It is important that this product be compatible with existing software. The UNIX program *Make* resembles this project but lacks many features which will be implemented. As such, it is important to be compatible with *Make*. An option will be provided to load an existing *Makefile* which will then be converted into the native format. In this way backward-compatibility will be maintained.

When a user chooses to be given a compiled or executable version of package the program will search the dependency tree to determine which files must be recompiled and which are current. In this way, a large software project can be easily maintained while only modified code will be recompiled. This can be crucial in a situation where a large software team is working together on a project and compile times could be tedious or burdensome on the network or workstation. Should a program have a specific design document as its dependant and the document be of newer date than the source code, it can be assumed that the designer (not the programmer) has modified the specification. The user will be notified of this and should then take it upon themselves to conform to the new specifications.

Unlike other configuration management systems, this product will not be aimed at a specific programming language. Therefore the user will be able to specify a particular compiler and compiler options. In this way the product will work with an arbitrary programming language, even those not yet available or conceived.

The user will be able to specify a directory structure to place their source and object or class files. This is an added feature that will facilitate the organization of the project.

Finally, the project will be able to be saved. In this way, the user will have to specify the compiler, compiler options, dependencies between files, and other project options only once.

A.2.2 Network

In order to accomplish the goal of allowing software components to exist anywhere on the Internet, the Software Configuration Management system will incorporate several remote/network features. These features will be extremely useful when different portions of a project are stored on several different servers. The system will automatically acquire any updated files from their remote locations in order to compile the entire system on a local server.

As the dependency tree is traversed, each of the date stamps on the remote system files are queried to determine the need for recompilation following the compilation order rules of Ada. If recompilation is deemed necessary, the remote source file is copied to the local host using any of the available user-determined means, such as *rlogin*, FTP, or HTTP. The source file is then compiled locally and the object file is stored in a pre-selected directory. Finally, in order to avoid redundancy and source file conflicts, the source file is deleted from the local system.

These remote / network features will necessitate the ability to interact with external hosts using several methods. Of course, several fields will be required by the system for a successful interaction, such as *userids*, *passwords*, and remote addresses.

The Configuration Management system will contain the functionality to rlogin to a remote host by allowing the user to specify the remote host, the userid, and the password required to obtain files from the remote host if it is determined that recompilation is necessary. In the case of a trusted host, the userid and password may not be necessary.

The Configuration Management system will have the additional functionality to FTP to a server and transfer the necessary source files to the local host for recompilation, providing the necessary login and password exchange.

The Configuration Management system will have the ability to access Internet files via an HTTP connection if the need arises.

Any of these options may be selected by the user via the graphical user interface while entering the initial dependencies.

A.3 Graphical User Interface

Java will be used to construct the Graphical User Interface. This will accommodate platform independence, as well as employ software reuse by way of the Abstract Windowing Toolkit and provide enhanced user-friendliness.

A graphic interface will greatly enhance usability by making software functions readily available to the user. Typical command line applications tend to overwhelm a new user with the arcane details of available options. A graphical interface will serve to decrease the learning curve for the new user and help an experienced user attain efficiency. It will also add stability by allowing control over program input and output.

The graphic interface will aid the user in creating, verifying, and maintaining file interdependence.

A.4 Paper

An important feature of the final paper will be to introduce and analyze some of the existing works with Configuration Management systems. From the technical report *Spectrum of Functionality in Configuration Management Systems* by Susan Dart, it can be noted that some component features that currently exist are RCS (Revision Control System) and DMS (Distributed Management System). RCS is a version management system and DMS provides version management for files distributed on different platforms.

Using current research, it will be shown why there is a growing need for Configuration Management systems. Many companies and software teams are currently creating projects which are distributed throughout the network and Internet. These projects are too large to be managed by a single user. Therefore the need arises for complex software to facilitate in this task.

A detailed description of how *make*'s capabilities will be expanded and what new features will be offered in the product will be provided. Some of the new features of the product will be that it will work with an existing *Makefile*, allow for network access to remote files, and provide a graphical interface for creating a dependency list.

Finally, the paper will include a formal specification for the product which will formally specify the important and vital operations.

A.5 Properties

A documentation will be provided in the end product, which explains installation of the product, provides test cases, and explains how the product works.

The management system will be self-realizable in that upon successful completion of the product it will be used to manage itself. By this it is meant that the final product will be executed and the project to manage using the configuration management tool will be the code for itself. This will then be used to produce the same product that is executing.

Portability will be accomplished through the use of Java as the chief programming language for the project. Java is accepted industry-wide as a language that is portable with Sun's 'Write Once, Run Anywhere' goal. Even though the product will not be invoking *Make*, it is foreseen that the product will use other shell commands of the standard UNIX environment. At this time it cannot be insured that the system will be portable to operating systems other than UNIX. With most of program development being done on UNIX based machines, this would appear to be a satisfactory limitation.

The product will employ software reuse through the user of Java's Abstract Windowing Toolkit. This will allow for a system independent and portable graphical interface. With the compatibility of *Make*, the system will adopt much of *make*'s functionality as well as the compilation order of Ada.

Efficiency will be maintained through well-defined and stable graphics provided by Java. Java is known to be slower than most other languages, but this is the cost for portability, which is seen to be of higher value.

The product will not target any specific programming language. This will allow for the use of different programming paradigms such as the imperative *C*, the object-oriented *Java*, or other paradigms and their languages as long as the languages support a dependency between programs and files. This is generally supported by the high-level languages through a linking stage of a compiler or through class interfaces.

A.6 Conclusion

The software configuration management system will greatly aid any company or software team in the development of their product. This extension of common configuration management packages will expand on existing programs and take advantage of the growing use of the Internet or local network for program development. The final product can be used by the coders or as high up the development ladder as the software designer who specifies the interfaces.

Appendix B

URL Class Example

What follows is an example implementation of the network specification specified in Section 4.2. This implementation allows for retrieval of anonymous FTP files, HTTP files, and files through the FILE protocol. The code is written in Java to allow for portability and is presented here simply as an example of how the implementation could work.

```
import java.net.*;
import java.io.*;
import java.util.Date;

public class GetURL {

  // Gets the date of the URL specified.  A null is returned if there was
  // a connection error.
  public Date getDate(String someURL) {
    URLConnection c;
    try {
      URL url = new URL(someURL);
      c = url.openConnection();
      c.connect();
    }
    catch(Exception e) {
      System.err.println(e);
      return null;
    }
    return new Date(c.getLastModified());
  }

  // Gets fileName from the URL specified.  If an error occurs, a -1
  // is returned, else a 1.
  public int getFile(String someURL, String fileName) {
    URLConnection c;
    try {
      URL url = new URL(someURL);
      c = url.openConnection();
      c.connect();
    }
    catch(Exception e) {
      System.err.println(e);
      return -1;
    }

    try {
      InputStream in = c.getURL().openStream();
      OutputStream out = new FileOutputStream(fileName);
      byte[] buffer = new byte[4096];
      int bytes_read;
      while((bytes_read = in.read(buffer)) != -1)
        out.write(buffer, 0, bytes_read);
      in.close();
      out.close();
    }
    catch(Exception e) {
      System.err.println(e);
      return -1;
    }
    return 1;
  }
}
```

A test run of this class is provided next. In this example three files are retrieved: one through HTTP, one through FILE, and the last through an anonymous FTP connection.

```
Script started on Wed Jun  3 06:08:03 1998
abyss(1)% pwd
/parl/ndebard/453/code/testing
abyss(2)% ls
GetURL.class   TestURL.class  typescript
abyss(3)% java TestURL
http://www.parl.eng.clemson.edu/~ndebard/index.html index.html
Connecting to URL : http://www.parl.eng.clemson.edu/~ndebard/index.html
Date of file      : Tue Mar 03 08:53:32 EST 1998
Saving file as    : index.html
abyss(4)% ls
GetURL.class   TestURL.class  index.html     typescript
abyss(5)% java TestURL file:/parl/ndebard/.xinitrc xinitrc
```

```
Connecting to URL : file:/parl/ndebard/.xinitrc
Date of file      : Wed Dec 31 19:00:00 EST 1969
Saving file as    : xinitrc
abyss(6)% ls
GetURL.class   TestURL.class  index.html     typescript     xinitrc
abyss(7)% java TestURL ftp://shredder.parl.eng.clemson.edu/etc/passwd passwd
Connecting to URL : ftp://shredder.parl.eng.clemson.edu/etc/passwd
Date of file      : Wed Dec 31 19:00:00 EST 1969
Saving file as    : passwd
abyss(8)% ls
GetURL.class   index.html     typescript
TestURL.class  passwd         xinitrc
abyss(9)%

Script done on Wed Jun  3 06:10:22 1998
```

Appendix C

Definitions and Descriptions

node Figure 3.2 is completely made up of nodes. These nodes can be thought of as objects, or points on a dependency graph. Nodes maintain a relation between their children (the nodes underneath them that they connect to) and their parents (those nodes that connect to them from above). Nodes do not have any innate properties but are a shell to be expanded upon by other types of nodes.

goal node In Figure 3.2 the circles are considered goal nodes. These are an extension of a node, which means that they know who their parents and children are. However, a goal node has other properties as well. A goal can have associated with it a command which is executed if it is determined that the node must be updated. This is most often useful when a goal has associated with it a command to compile all of its children files.

file node In Figure 3.2 the rectangles are file nodes. File nodes cannot have commands associated with them but are an extension of a node. File nodes have location on the network. This provides for the basic element of the dependency tree of a software project. Because a file node actually exists somewhere, it also has associated with it a datestamp.

datestamp A datestamp is a time and a date. Each file node has associated with it a datestamp so that the system knows the last date when it believes the file was last modified. The purpose of this is so that this datestamp can be checked against the date of the file returned through the standard POSIX interface. In this way, the system can determine if a file has changed and therefore needs to be updated. This information is then used to replace the old datestamp associated with the file.

Bibliography

[Midha] A. K. Midha, "Software Configuration Management for the 21st Century," *Bell Labs Technical Journal.*, Winter 1997, pp. 154-165.

[Salamone] S. Salamone, "More Control, Fewer Headaches," *Byte.*, January 1996, pp. 159-160.

[Dart] S. Dart, "Spectrum of Functionality in Configuration Management Systems," Carnegie Mellon University, *http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html*, December 1990.

[Bersoff] E. H. Bersoff and A. M. Davis, "Impacts of Life Cycle Models on Software Configuration Management," *Communications of the ACM.*, August 1991, pp. 104-118.

[Buckley] F. J. Buckley, "Implementing a Software Configuration Management Environment," *Computer.*, February 1994, pp. 56-61.

[Cronk] R. D. Cronk, "Get Control of Cross-Platform Development," *Datamation.*, August 1993, pp. 45-46.