

Site

A Language and System for Configuring Many Computers as One Computing Site

Bent Hagemark

Brown University
Providence, RI 02912

April 30, 1990

Abstract

This work describes the design and implementation of a language and system for use in maintaining the configuration of many computers as one computing site.

Introduction

A modern computing site typically consists of many computers networked together. This represents the next step in the evolution from large central time sharing computers accessed via many terminals. Among the problems encountered in ensuring that a site of many networked computers works as intended is the problem of maintaining an accurate and consistent configuration of the computers in the site. There are few solutions which address the broad scope of specifying configuration information on a network-wide basis in a site where many computers which must cooperate with each other and with common resources. This work presents a solution to the specification problem for the configuration of the subsystems found on a typical site consisting of networked UNIX workstations and servers. A simple utility demonstrates the use of this specification language as a basis for automating maintenance of site configuration. This tool is intended, for use by computer novices as well as expert systems programmers.

Organization of This Thesis

This paper consists of 4 chapters and an appendix.

Chapter 1 introduces and motivates the problem.

Chapter 2 outlines the framework of the solution.

Chapter 3 describes other approaches to solving the problem.

Chapter 4 is a detailed description of the implementation of the system.

The Appendix is the User's Guide to the system.

Contents

Chapter 1: Statement of the Problem	1
The Problem	1
Configuring a Large Site	1
Config Files in a Large Site	10
Why Solve This Problem	11
Chapter 2: Framework of the Solution	13
The Solution	13
Character of the Solution	13
Use of the Site System	16
The 3 Conceptual Levels of a Sitefile	18
Using Sitefiles	20
Pulling the Levels Together	27
Object Oriented	32
Theory of the Site System	34
Future Work	38
Chapter 3: Other Attempts	40
rdist	40
make	43
Interactive Programs	43
config	44
YP	44
Chapter 4: Implementation and Reference	46

Site 1.0 Implementation	46
Site 1.0 Modes of Operation	46
Site 1.0 Low Level Modules	50
Sitefile Grammar	62
Appendix: User's Manual	65
Using Site 1.0	65
How To Prepare a Sitefile	65
How to Write a Driver	67
Invoking the <i>site</i> command	73

Bibliography

- [1] *Chapter 14: The Sun Yellow Pages Service*, System and Administration Guide, SunOS 4.0, Sun Microsystems, May 1988.
- [2] *make(1)*, 4.3BSD UNIX Users's Reference Manual, University of California, Berkeley, California, April 1986.
- [3] *rdist(1)*, 4.3BSD UNIX Users's Reference Manual, University of California, Berkeley, California, April 1986.
- [4] S. J. Leffler and M. J. Karels, *Building 4.3BSD Unix Systems with Config*, 4.3BSD UNIX System Manager's Manual, University of California, Berkeley, California, April 1986.

CHAPTER 1

Statement of the Problem

1.1. The Problem

Managing the configuration of many computers which together form a single computing site is the problem addressed in this work. This section motivates the problem using UNIX workstations and servers as an example.

Terminology

To help avoid confusion about terms often used to describe a computing facility we define how the following terms are used in this work.

computer A single self-contained workstation, server or traditional computer. A part of a site.

installation This term refers to the activity of setting up a computer or site.

network The physical local area or long distance methods for connecting computers to each other. Not all computers on a particular network necessarily belong to the same site.

site One or more computers and peripherals all under the same ownership or administrative domain. The computers and peripherals of a site are often interconnected with one or more networks.

1.2. Configuring a Large Site of Many Computers Running UNIX

The following sections describe some of the activities one must perform to configure a computer running UNIX and how to configure each computer for common site-wide services and resources. The "activities" of configuring a site are mostly those of editing a lot of files on a lot of computers. The following sections show that editing a lot of files on a lot of computers is tedious and error prone when done by human hand, but rote enough to suggest an automated solution.

The Workstation Refinement of Time-Sharing

Our primary concern is with the configuration needs of a large site (on the order of 100's of computers) under centralized administrative control. We further assume that the intent is to configure a site which the user's view of the system is similar to a large computer which they accessed using serial line terminals. This *workstation refinement of timesharing* involves configuring a set of servers and client workstations to provide the same degree of transparency as that of timesharing systems in the past. That is, the user sees the same home directory, and the same site-wide services and resources such as printers, mail, and news service no matter which machine uses. The user also plays no role in administering "their" machine in such an environment. It's not our feeling that a user should be prevented from doing so more than we feel that a user shouldn't *have* to.

Basic Systems Configuration

The following sections are presented to provide the reader with a common basis for understanding what problems we trying to solve (and which problems we are *not* trying to solve!). They do not pretend to be an exhaustive guide to the systems software found on actual UNIX workstations.

On each computer the administrator must prepare a variety of files to configure each aspect of the computer. These are files which specify the basic configuration of the system itself (i.e., the peripheral devices such as displays, mice, keyboards, serial lines for terminals, tape drives, disk drives and network interfaces and file systems.) In addition one must configure the system for site-wide services which include network routing, network name service lookup, remote printers, electronic mail, and distributed file systems.

We will refer to any file used in the configuration of a subsystem on a computer as a *config file*. Figure 1.1 lists the names of various config files found on a computer running UNIX. (We will be using the Berkeley UNIX derivative of SunOS for the examples in this work. Refer to the UNIX Systems Administrators Guide or the appropriate manual pages for more complete information about configuring a UNIX or SunOS system).

Config file name	Subsystem
/etc/fstab	file systems
/etc/exports	NFS
kernel config file	devices
/etc/printcap	printers
/etc/sendmail.cf	mail
/etc/ttytab	terminals
/etc/hosts	networking

Figure 1.1: Some UNIX config files

When configuring a single machine or a small number of machines in a relatively quiescent site it is relatively straight forward to edit these files directly. When one faces many 10's, 100's, or 1000's of computers it is no longer reasonable to manage each machine by editing the resulting 100's, 1000's, or 10,000's of files.

The following paragraphs discuss the more important and interesting subsystems and their configuration requirements. It is very important to notice how single pieces of information become part of many different config files. Our focus is on the *information* needed to produce each config file and where this information comes from. We highlight information used in many different config files on the same computer as well as information used in the same config file on many different computers.

Disks and Filesystems

Each computer in a site usually has its own disk drive. Although diskless machines differ at a lower level they have roughly the same configuration needs as a machine with disks at the level of discussion in these sections.

One aspect of configuring a disk drive for use on a UNIX system is dividing the disk into several *partitions* which are either used as paging devices or as *file systems*. The /etc/fstab file describes this information. See Figure 1.2 for an example of the contents of this file.

```
#
# /etc/fstab
#
# Disk devices:
#
/dev/sd0a / 4.2 rw,nosuid 1 1
/dev/sd0b swap swap rw 0 0
/dev/sd0g /usr 4.2 rw 1 2
/dev/sd0d /tmp 4.2 rw 1 3
/dev/sd0e /aux 4.2 rw 1 4
```

Figure 1.2: Contents of /etc/fstab file

Each line of the file describes an association between a disk partition (/dev/sd0g) and a file system directory name (/usr). In this case the "sd0" part of the disk partition name means "SCSI disk, unit 0". The "g" is a slice or partition of that disk unit. Some other aspect of configuring the disk drive defines exactly which physical tracks and sectors are in this partition. When the UNIX system boots up the mount command takes each line of this file as input and tells the UNIX kernel which partition of the disk to access for files in (and below) these directories. For example, any file name starting with the name /usr (such as /usr/people/bh) resides on the disk partition /dev/sd0g; the file /tmp/foo/ is found on the /dev/sd0d partition. Adding more disks to the system requires additions to this file. Changing the layout of the disk -- changing the sizes or locations of the disk partitions -- may involve changes to this file.

The cases covered so far pertain only to the local system. That is, the contents of the /etc/fstab file depends only on information about the computer on which it is located. In summary this information includes:

- disk devices and partitions, and
- file system directories.

The following section describes information placed in the /etc/fstab file to configure distributed file systems.

Network File Systems (NFS)

The `/etc/fstab` file also describes how remote file systems are configured. The `/etc/fstab` also describes mappings of directories on the local machine to directories on other computers. In Figure 1.3 are some example lines from an `/etc/fstab` configured for NFS. As with local file systems the second word in the line refers to the name of the directory on the local system. The first word in this case takes the form `<hostname>:<directory>`. For example, this file indicates that the files in and below the directory `/pro` are physically located in the directory `/pro` on the server named `garcon`, and that files in `/home/ober` are on `ober` in its `/home` directory. The remaining information on each line is a set of parameters used by the NFS subsystem. We won't describe what they mean except to note that these parameters are not dependent on information outside the host itself and there is generally no requirement that these parameters need to be consistent with information on other systems.

```
#
# NFS directories
#
doorknob:/var/spool/mail /var/spool/mail nfs rw,hard,bg,intr,timeo=14 0 0
garcon:/pro    /pro          nfs rw,hard,bg,intr,timeo=28 0 0
garcon:/home   /home/garcon   nfs rw,hard,bg,intr,timeo=28 0 0
ober:/home     /home/ober     nfs rw,hard,bg,intr,timeo=28 0 0
ober:/map      /map           nfs rw,hard,bg,intr,timeo=28 0 0
```

Figure 1.3: NFS client configuration in `/etc/fstab`

This configuration information is our first example of information which needs to be consistent between 2 or more separate computers in the site. The administrator of the local host is free to decide independently where the remote NFS directories should be mounted. But, the information about remote file server names and the names of directories on those machines obviously depend on how those remote systems are configured.

An NFS server must configure which directories its making available to other machines. This information is found in the `/etc/exports` file on the server. (Typically, the presence of an `/etc/exports` file defines a system as a file server). See Figure 1.4 for the contents of an example `/etc/exports` file. Each line of this file defines a directory which can be exported (NFS-mounted on a remote machine)

and the names of machines which are permitted to do so.

```
#  
# /etc/exports for machine garcon  
#  
/pro -access=bob  
/home/garcon -access=bob,mary
```

Figure 1.4: NFS server configuration

For a machine to properly access files remotely over NFS the information found in its own `/etc/fstab` must be consistent with the information found in each server's `/etc/exports` file. Adding a disk to a file server or a whole new file server requires updating the `/etc/fstab` and `/etc/exports` file on the server and the `/etc/fstab` of each computer in the site.

NFS involves a number of pieces of information which must be consistent between several machines and between several files:

- NFS server host names,
- NFS server directory names,
- NFS client host names.

Furthermore, an NFS file server must keep information consistent between its `/etc/fstab` and `/etc/exports` files. Any directory exported by the server must be located on a disk device on the server. This information includes:

- access permissions,
- server hostname
- file system directories

Network Name Service

Each computer on a network is known by a *hostname*. Most network-based subsystems require consistency in the host name lookup mechanism in that each host must agree on the mappings between host names and host network addresses. For example, clients and servers need to know about each

other to share file systems over NFS (as described in the above section).

Hostname and network number information is stored in an `/etc/hosts` file which is distributed to each computer, or through a network distributed database such as YP or BIND. However, even with YP each computer must have its own `/etc/hosts` with an entry for at least itself so that the boot up process can start up the YP service! From the perspective of an administrator taking care of a large number of computers in a site it's usually more difficult to distribute to each computer a unique file for each than it is to distribute the same file to each. In any event the information about a mapping from hostname to appears in the `/etc/hosts` file as in Figure 1.5.

```
#  
# /etc/hosts  
#  
128.148.32.34      barney  
128.148.32.56      garcon  
128.148.32.93      doorknob  
128.148.32.45      ober
```

Figure 1.5: hosts file entries

Printers, spool directories and `/etc/printcap`

The Berkeley UNIX line printer spooling system provides site-wide access to printers. Under this system each computer in the site is configured for access to printers connected directly to the computer or to those connected to other computers in the site. Whether a printer is local or remote can be hidden from the user by configuring the print system appropriately. That is, the user can always print to the same printer and know it by the same name from any host in the site if the printer subsystem configuration information is consistent throughout the site.

```
#  
# /etc/printcap  
#  
ps0:sd=/usr/spool/lpd/ps0:if=/usr/local/lib/pscomm:lp=/dev/ps0  
ps1:sd=/usr/spool/lpd/ps1:if=/usr/local/lib/pscomm:lp=/dev/ps1
```

Figure 1.6: /etc/printcap entries on a print server

The printer system is configured on each host -- whether it has a printer connected directly to it or not -- by making entries in the `/etc/printcap` file. On a host with its own printers the `/etc/printcap` file describes the names of the printers and the various print filters supported by each printer. See Figure 1.6 for an example of this case. Each line in Figure 1.6 configures a printer. The first word in the line before the ':' is the name of the entry (and usually the name given to the printer). The 'sd=' field in the entry gives the name of the spooling directory, the 'if=' entry gives the name of the print filter and the 'lp=' entry specifies the device file corresponding to the printer.

```
#  
# /etc/printcap  
#  
ps0:sd=/usr/spool/lpd/ps0:rp=ps0:rm=kclner  
ps1:sd=/usr/spool/lpd/ps1:rp=ps1:rm=kclner
```

Figure 1.7: /etc/printcap entries on a print client

On hosts wishing to access to printers on other hosts the `/etc/printcap` file describes the host-name and printer name as appropriate. See Figure 1.7 for an example `/etc/printcap` for a print client.

Adding a new printer involves adding an entry to the print server's `/etc/printcap` and an entry in the `/etc/printcap` file on each host in the site. That is, each host in the site depends on information about each print server in the site. The following list summarizes the configuration information needed in the print spooling system.

- printer name
- remote printer name and hostname of print server

- printer device
- print filter(s)

Unlike the `/etc/hosts` file where the same file can be put on each system, here the file for a machine that has a printer locally is different than the file on a machine that uses it remotely.

Kernel

The kernel is the operating system executable image. The bootup process of the computer involves reading this in and running it.

Production of the BSD UNIX kernel itself is handled by *config*. There are several pieces of information which must be consistent between the kernel and other subsystems for a computer to work properly.

The `/dev/MAKEDEV` script file automates the production device files. The major and minor device numbers of device files (usually found in `/dev`) must be consistent with information in data structures found in the UNIX kernel.

The kernel *config* file must be consistent with the `/etc/fstab` file for information about the root file system and swapping/paging device.

The following lists some of the information needed to configure the kernel:

- cpu type,
- disk drive(s),
- tape drive(s),
- network interface(s),
- display device(s).

The important point here is that the kernel config file depends only on the local machine. This differs from the `/etc/hosts/IP` and `/etc/printcap` files which depend on information about other computers in the site.

User accounts, home directories, `/etc/passwd`, `/etc/group`

Information about the people who use the site are in a way part of the site's configuration. A user is defined by an entry in the `/etc/passwd` file (or YP `passwd` map) and possibly several entries in the `/etc/group` file or YP map. The information associated with a user includes the user's login name, the user's uid and gid names and numbers, their real name, office, and phone number, their home directory and the login shell. Some sites "hide" the server or file system location of user's home directories using symbolic links and NFS mounts which adds to the information about a given user. There is information inherent in these symbolic links or NFS mount which by nature is usually not in the `/etc/passwd` file.

There are also users which don't represent people. These users identify processes and files of various subsystems. The print spooling server and user access processes run as the user `lpd` and the spool directories are owned by `lpd`. The user `lpd` must be defined in the password file or YP password map for the print spooling software to operate properly.

1.3. Problems With Config Files in a Large Site

The administrator of a large site must maintain many config files on many computers. Some of these files are unique to each computer and some are the same on each. Some files depend on information about other computers. For example, the configuration of NFS and printer system clients depends on information about NFS and print servers. Adding a file server, adding a disk to a file server, or adding new printer often requires editing at least one file per machine in the site in addition to editing some files on the machine directly affected.

Additionally, there are a wide variety of syntactic demands each subsystem imposes on the associated config files. Even the smallest syntax error can lead to disastrous consequences often with little or no diagnostic warning about the problem being as simple as a syntax error in a file.

Even though there are tools for performing systems administration activities on individual machines there are few tools for dealing with many machines which together form a site. Current tools often can bring a single machine up and running on its own to the point where it's usable as an

individual system. But in sites managed in the transparent fashion we discussed at the outset this is only a small part of the initial configuration process. After the machine's basic installation is complete (loading software or doing diskless configuration on the server side) the administrator must configure the machine for services specific to that site.

Tools and systems to help automate the initial installation of a machine AND the initial configuration and customization for site-specific services are needed. It should be straight-forward for a machine new to the site to configure itself on bootup based on site information it retrieves through a well-known location serviced elsewhere on the network.

There exists no formal method for documenting to users (or other administrators for that matter!) any site-specific configuration, customization, and tailoring information. There are manual pages for describing how to use various network services, but rarely is the information about printer servers, and file servers presented in the same form. Documentation about the site also depends on information about the configuration of the site. For example, most users do not know to find or read the `/etc/printcap` file to determine which printers exist.

Service technicians and customer support people also need to understand a site's configuration. A person in this position is often less familiar with the site and its various parts than an administrator which works with the site daily and thus must piece together the bigger picture by perusing many different config files on many machines before they can be of help in solving a problem with the site.

1.4. Why Solve This Problem

Maintaining the configuration of a large site requires keeping track of a many configuration files on many computers. The information about the site inherent in these files is spread site-wide with many individual pieces of information repeated.

This is an important problem to solve as inconsistencies between configuration files can lead to failure of parts or or even the whole site. These files additionally have stringent format requirements. Thus, a large site often demands the attention of a "guru" or "wizard" merely to keep it configured properly even though the repetitious nature of this work suggests this could be automated. As

networks of workstations and even personal computers become more popular and replace or at least augment traditional centralized time-sharing and mainframe environments there is greater need for easy to use systems management tools. The solution presented in the remainder of this work sets the groundwork for automating the configuration aspects of managing such a site.

CHAPTER 2

Framework of the Solution

2.1. The Solution

This chapter describes the **Site** system and the **Sitefile** which we use to solve the problems described in Chapter 1.

The solution takes a step back from the configuration files themselves and focuses on managing the information inherent in configuration files - the goal being to automate the production and maintenance of configuration files. The key design feature in automating this process is to generate all configuration files from a single common representation. Figure 2.1 illustrates the solution.

We consider the specification problem the issue of greatest importance in automating this aspect of site administration. The visual appearance and related user interface issues and the technical details of a robust and secure networking implementation for distributing configuration information are each separate problems and outside the scope of this work.

Character of the Solution

The **Site** system design reflects 1) the activities performed in configuring a site AND the people who perform them, and 2) the information needed to configure the site. **Site** is based on a design which includes a philosophy of who should be doing what activities as well as on the lower level concerns of the most "natural" and effective way to describe a computing site and its various pieces. The system is designed to be easiest to use for those activities which are performed most often, yet powerful enough to be used by expert systems programmers to describe low-level customization and modification of the systems software. The key is that one would like to and *can* decouple the two activities and cover most of the configuration needs of a site in the former case.

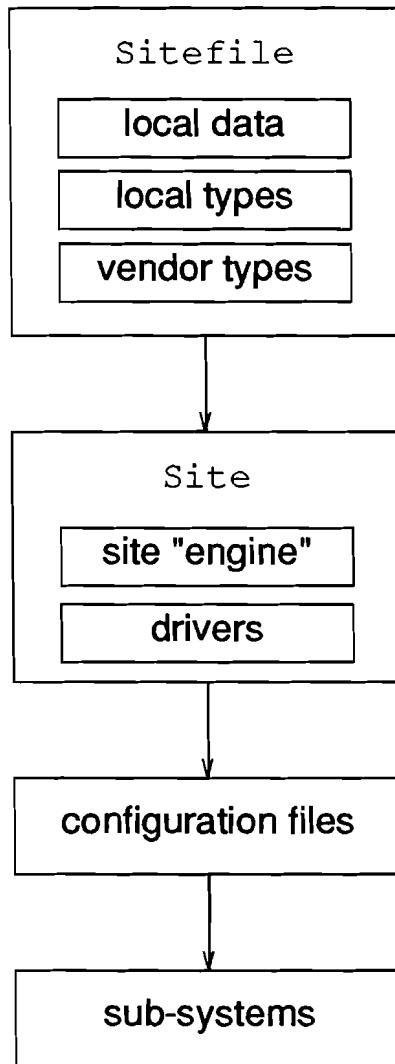


Figure 2.1: Framework of the solution

Our experience with modern UNIX systems administration is that it is mostly an organizational and administrative task. In fact it is often the case that some knowledge may be more dangerous than little knowledge! This is a departure from UNIX systems administration of the past where "kernel hacking" and related systems programming activities were considered an immutable part of managing the site (a "site" of probably only a single mini computer). In modern world of commercial UNIX systems one finds a drastically more complex site than this single machine of the past. The site now includes many machines potentially from several different vendors, and each machine is several times

more complex than the original machines UNIX ran on. It is no longer possible to be a "guru" for *every* aspect of the system any more. The final point is UNIX users are increasingly "real" people who don't *care* how the machine is set up. They are interested in using the system for various applications. We do not intend to satisfy the UNIX "guru" -- at least not directly. Indirectly we hope to satisfy the "guru" by relieving such a person from some set of mundane tasks.

The next most important set of design goals are to describe the site as a whole effectively for both people and software tools. A central database describes the site from a conceptual level. The information in the database is stored independently from the configuration files and is the source of the data required to generate the configuration files. The database uses a text file "mini-language" which in a declarative style using object oriented data abstraction with multiple inheritance. The central database is not merely a central repository of configuration files.

Finally, the design permits an efficient network-based implementation. Since the Sitefile and Site system are intended for use on large sites it is important that the processing involved can be distributed. In this chapter we describe the higher level design which permits this. In Chapter 4 we present the actual implementation details.

The following list summarizes the design goals and features roughly in order of importance.

-
- target different config activities at appropriate people
 - automate as much as possible while allowing for extensibility and customization
 - lean and mean "tools" approach to technical implementation
 - provide a precise, formal, and natural description of the site
 - organize information about the site, NOT the configuration files in the site
 - declarative data description style
 - object oriented: hierarchical typing with multiple inheritance
 - efficient client-server network implementation
-

2.2. Use of the Site System

The Site system consists of the *site* program which processes a *Sitefile*. A Sitefile is the central database representing all information about the site. The multi-level use of a Sitefile is the most significant aspect of this solution and is the subject of most of the discussion in this part of the Chapter. (Technically speaking a Sitefile doesn't really have 3 sections. We just use it that way).

This system identifies and separates 3 levels of understanding and activity in the configuration of a large computing site. The top level includes the activities of adding, moving, and changing computers, networks, and printers. The information needed to do so is conceptually easy to understand and most importantly, requires little programming expertise or knowledge of how things are implemented. The middle level is concerned with an overall site-wide view of the design of the site and its services as a whole. This involves policy decisions of issues such as network topology and network name service. This sort of activity is less frequent than that of making small additions or changes to the site. Working at this level requires some knowledge of how the site and the parts of the site work. The information at this level is logically disjoint from the top level. The bottom level presents information about systems internals which may be necessary in configuring the site. This level differs from the middle level in that it does not try to group parts of a site together. This level of the system describes what parts are available while the middle level describes how those parts are used at a given site. For example, this level may describe several types of network name service leaving the decision of which to use on what machines to the middle level. This level includes descriptions of "vanilla" software and hardware products from vendors.

The goal of targeting different configuration activities at the appropriate people is realized as follows. In good information-hiding practice the mechanisms below a given level are automated. At the top level we target the non-programmer whose work is carried out by automation at the middle level. At the middle level we target a site's systems programmer who works in terms of the lowest level. The lowest level is aimed at the computer vendor's engineering staff who provide the hooks to the specifics of their underlying system.

The Site Program

The Site system consists of a simple utility which takes a Sitefile as input and outputs all configuration files for all the computers in the site. The program Site is a "configuration file compiler" of sorts with the format of the Sitefile being the "language".

Site Client-Server Model

In order to permit efficient and scalable network-based use the Site system implementation defines an internal query protocol. The protocol is based on splitting the processing of a Sitefile into client and server parts. The most scalable network implementation places as much processing as possible on the client side. The possible interdependency of information between any set of objects in the site leaves the server side with the task of reading and parsing the Sitefile. However, after the Sitefile has been parsed one may produce the config files for each computer in the site in parallel. The best place for this processing is on each computer itself. The intended use of this split is for each computer in the network to periodically (daily, and at system start up, for example) contact the Site server to retrieve the site information needed to produce the correct configuration files. This "pull" model is more efficient than a "push" model whereby the Site server (sequentially) produces all config files for all computer and then distributes them. The details of the protocol and client-server implementation are described in Chapter 4.

We talk more about the Site program, the Site client-server model and protocol, Site system implementation, and use in Chapter 3 and the Appendix.

2.3. The 3 Conceptual Levels of a Sitefile

Figure 2.2 summarizes our view of how site configuration activity and information is best organized. We divide the Sitefile into the Local Data area, the Local Types area, and the Vendor Types area.

At the **Local Data** level are the data directly specifying the inventory of the site. This data is conceptually the easiest to understand and manipulate and also is the type of configuration data which sees the most activity.

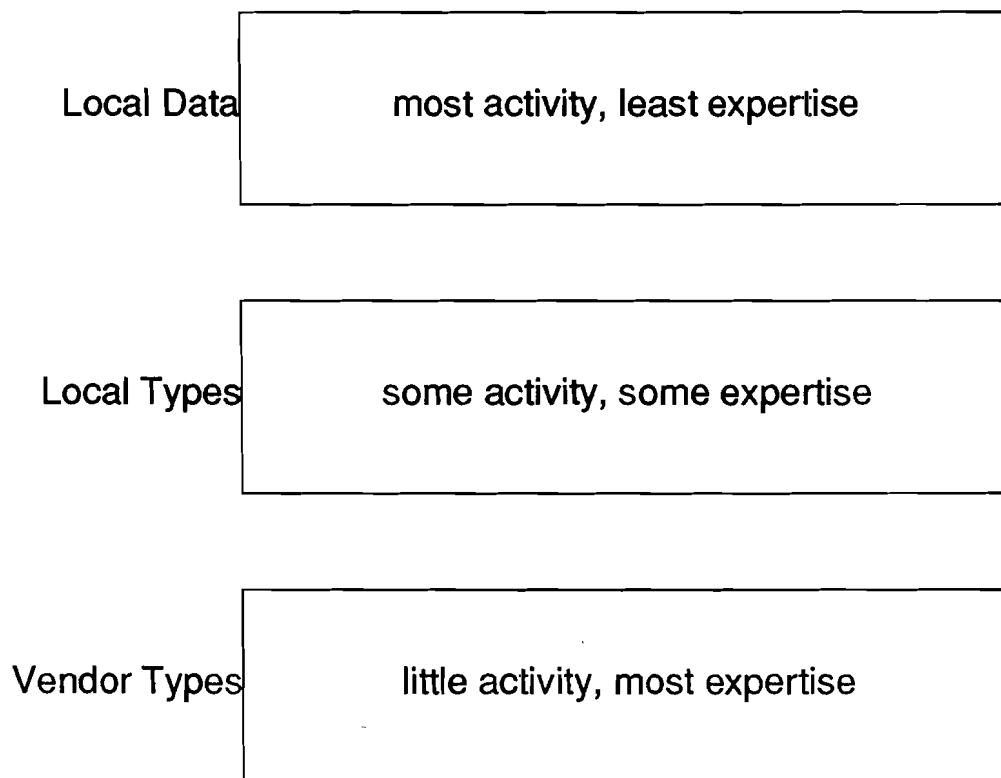


Figure 2.2: Multi-Level Solution

For example, the actions involved in adding a new computer of a previously existing type takes place at this level and *only* at this level. This only involves Sitefile manipulation analogous to filling in a few blanks in a form.

The **Local Types** level contains the data specifying site-wide policy and other information specific to the site as a whole, but not directly associated with any particular object in the site. This information is more conceptually difficult to understand and manipulate as the information here is mostly of local classification of the way a site sets up a vendor's systems. To work at this level requires some computer programming expertise.

For example, adding a new type of machine to the site first may involve some work at this level for the installation of the first such new machine. Future similar machines should only involve making a few Sitefile entries in the Local Data area. While work at the Local Data area is pretty much filling in blanks, the work at the Local Types area is specifying what the blanks should be.

The **Vendor Types** level contains information from the vendors supplying the systems to the site. This information includes default configuration information for the systems listed in a vendor's price book, for example. Ideally this information should provide reasonable defaults for that vendor's hardware and software systems which would result in a configuration either most recommended by the vendor or at least one which provides a usable system. These defaults may stand or may be overridden by the site's administration. Any such over-riding belongs in the Local Types area.

The following sections examine each level in greater detail. This is followed by discussion of how the levels are used along with some example configuration activities which further illustrate when and how things change at the different levels. We conclude with a discussion of the theoretical or conceptual basis of the solution.

2.4. Using Sitefiles

Here we present a detailed description of what appears at each level of the Sitefile.

The Top Level of a Sitefile: Local Data

The top level of a Sitefile directly describes the inventory of the site such as the workstations and servers, the networks, the printers, and any other such items that comprise the site. This level holds high-level information associated with these items such as its name, and what kind of item it is, and other information most administrators would most commonly know *and understand* about it. **The idea of separating the information describing parts of the site from how the information is used is the most important feature of this system.** This technique is certainly nothing new or novel. What is new and novel is applying this to configuring a large computing site.

It is also not new or novel to talk about systems and configurations in terms of types. In the "real world" of computers (the one where computer vendors sell systems to customers who put them together and make a site out of them) a sales person puts together a quote or bid listing the *types* of systems they can offer. These are listed in the vendor's price book in various tables and lists which the salesman understands. If a customer wants a Sun3/60 with 8 megabytes of memory, color display, and 140 megabyte disk the salesman consults his price book and responds with a name such as "Sun-3/60-S4". Even at the time of purchase of the computer there is discussion of types and even type names! Unfortunately this is also usually the last time someone in a site deals with a system having a type.

This top level of the Sitefile is known as the *Local Data* area. Figure 2.3 presents some example Sitefile entries from this area.

```

#
# Sitefile Local Data entries
#
BrownWs bob {
    hostname =      bob;
    netaddr =      128.148.32.42;
}

BrownPrinter ps0 {
    printname =     ps0;
    printserver =   kelner;
}

EtherNet lab33 {
    networkname =   lab33;
    networkaddr =   128.148.33;
    gateway =       garcon to net32;
    gateway =       garcon to default;
}

```

Figure 2.3: Sitefile Local Data Entries

Each Sitefile entry at the Local Data level is a Sitefile *variable* entry whose general form is as follows.

```

variable-name type-name {
    [field ;]
    .
    .
    .
}

```

As indicated by the square brackets a variable entry need not contain any fields. Also, newlines are not significant as semi-colons ";" separate fields from one another and the curly braces "{}" mark the start of the fields and the end of the whole variable entry.

A variable entry field can take one of two forms:

```

attribute-name = attribute-value ...
component-name := component-driver()

```

The first form is an *attribute* entry and the second form is a *component* entry. Note that an attribute's value may consist of one or more words. Also, note that we often refer to "variable entry"

or other varieties of "entry". This is to make clear the distinction between the syntactic form which appears in the file and the overall meaning and value a variable when considered along with its type. (More on types in the next section).

Given the basic syntax we present some example Local Data entries. See Figure 2.3 for some example entries as they would appear in a Sitefile. The first entry is the variable `bob` which is of the type `BrownWs`. The first attribute of `bob` is its hostname which has the value `bob`. Obviously this is intended to mean that this workstation's hostname is "bob"; however, it's important to note that the Sitefile itself has no idea what it *means* for a `BrownWs` to have a hostname. It merely stores the association of the variable `bob` having this thing called a hostname valued at `bob`. It is up to the reader of the Sitefile -- be that a human or a software tool -- to extract meaning from this association.

A variable entry may specify the same attribute name in more than one field. In the case of `lab33` it is meaningful to understand that the host `garcon` is the gateway from this network to two other networks.

The following summarizes the concepts behind the Local Data area of a Sitefile.

- The Local Data entries represent the list of items (objects!) comprising the site.
- Each entry describes high-level site-specific information about each item.

The Middle Level of a Sitefile: Local Types

The next lower level of a Sitefile describes site-specific policy of how various parts of the site are used and information about how the parts are most naturally organized at the site. This is known as the *Local Types* area of the Sitefile.

```

#
# Sitefile Local Type entries
#
class Printer {
    printname =      ;
    printserver =    ;
}

class BrownPrinter : Printer QmsPrinter {
}

class BrownWs : Sun-3/60-S4 YPClient {
    ypdomain =      ;
}

class EtherNet {
    networkname =    ;
    networkaddr =    ;
    netmask =        255.255.255.0;
    domain =         cs.brown.edu;
}

```

Figure 2.4: Sitefile Local Type Entries

Each Sitefile entry at the Local Types level is a Sitefile *type* entry whose general form is as follows.

```

class type-name [: base-type-name ...] {
    [field ;]
    .
    .
    .
}

```

As with variable entries a type entry need not contain any fields, and newlines are not significant. A type entry field is in general the same as a variable entry field, except that a type entry may specify attributes or components which have no value or driver, respectively. We refer to a value-less or driver-less entry as a *virtual* attribute or component. The meaning of such an entry is covered shortly. A type entry field takes one of the two forms:

```

attribute-name = [attribute-value ...]
component-name := [driver-name()]

```

Figure 2.4 presents some example entries found at this level. The first entry defines the type `Printer` as something which needs a `prntername` and a `prntserver`. (This entry does not indicate that the site necessarily has any printers nor does it imply that there are any objects in the site classified as a `Printer`.)

The second entry in Figure 2.4 defines the type `BrownPrinter` to be a combination of a `Printer` and a `QmsPrinter`. This introduces the Sitefile feature of **multiple inheritance** of type definitions. In this case the type `BrownPrinter` incorporates all of the fields from the types `Printer` and `QmsPrinter`. As presented in Figure 2.3 the variable `ps0` was defined to be of the type `BrownPrinter` where it was given the values of the `prntername` and `prntserver` fields. (The type `QmsPrinter` is defined in the lowest level of the Sitefile which is the subject of the next section).

The third entry defines the type `BrownWs` in terms of the types `Sun-3/60-S4` and `YPClient`. The actual definition of these types is defined in the lowest level of the Sitefile. The meaning at this level is that a workstation in this site is the same hardware configuration of a `Sun3/60` and is also configured at this site to be a client in the YP system. There are some important and powerful features in this simple syntax. As promised at the outset each level can "black box" anything going on below it. The point with the Local Types level is to isolate these site-specific "policy" and classification entries from the raw list of site inventory AND to isolate these entries from those produced by the vendor. (We've jumped the gun here a bit: the lowest level is known as the Vendor Types area with the intention that entries at this level be produced and maintained by the vendor's engineering staff and are distributed along with the systems software to the customer site where they are treated as "read only" and used as the basis for what we're discussing in this section). That is, entries at the Local Data level do not *need* to know that a `BrownWs` is either a `Sun3/60` or a YP client. A `BrownWs` still has a `hostname` and `networkaddr` and most importantly the people making entries at the Local Data level also do not need to know whether a `BrownWs` is getting its name service from YP or from its own `/etc/hosts` file or from something else such as BIND. Most importantly, though, is that someone working at the Local Data level does not have to know how name service lookup is implemented. (Consult a UNIX systems administrators guide for more information about network name service).

The features of hierarchical type definition in terms of multiple types is a powerful object-oriented technique which proves very useful and natural in organizing information about a computing site.

Before describing the next level of a Sitefile we summarize the concepts of the Local Type level.

- Local Type entries define site-specific policy and classification of site objects.
- Local Type entries are defined in terms of Vendor Types.
- The Local Type area is targeted at the site's systems programmer(s)

Depending on how much a site customizes and tailors a vendors systems it may have a very simple or very complex Local Type area.

The Lowest Level of a Sitefile: Vendor Types

The lowest level of a Sitefile describes vendor-specific classifications of a product line. Entries at this level are produced by the vendor as part of the same engineering effort which produces the system software itself. The vendor then distributes these entries which each site would then place in the area of the Sitefile known as the Vendor Types area. Hopefully, that this will more formally and accurately transfer the vendor's intent of how their systems should be pieced together out there in customer sites.

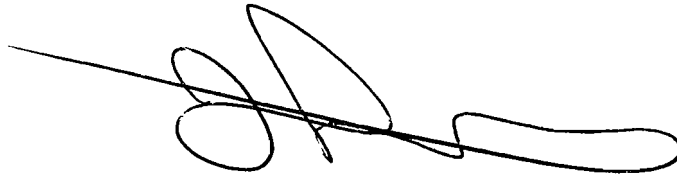
BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M4

Site
A Language and System for Configuring Many Computers as One Computing Site
by
Bent Hagemark

Site
A Language and System for Configuring Many Computers as One Computing Site
Master's Thesis

Bent Hagemark
Department of Computer Science
Brown University
April 30, 1990

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in the Department of Computer Science at Brown University

A handwritten signature in black ink, consisting of several loops and a long horizontal stroke at the end, positioned above a horizontal line.

Professor Kenneth Zadeck
Advisor

```
#
# Sitefile Vendor Type entries
#
class Sun-3/60-S4 {
    cpu =      MC68000;
    mem =      8 mb;
    disk =     140 mb scsi;
    device =   cg4;
    device =   le0;
    maxusers = 16;
    /vmunix =  SunKernel();
}

class YPClient {
    ypdomain = ;
    /etc/hosts := StubEtcHosts();
}

class YPServer {
    ypdomain = ;
    /etc/hosts := FullEtcHosts();
}

class QmsPrinter {
    filter =   if /usr/local/lib/ps/psif;
    filter =   of /usr/local/lib/ps/psof;
    filter =   df /usr/local/lib/ps/psdf;
}
```

Figure 2.5: Sitefile Vendor Type Entries

As with the Local Types level, the Vendor Types level consists of Sitefile type entries. The Sitefile type features discussed above apply here as well. Figure 2.5 illustrates the use of the *component* field. A component represents a config file. The name to the left of the "!=" is the config file name; the name to the right is the *driver* which is called in the processing of the Sitefile to produce the contents of the file. We present config file drivers in a later section. The salient feature of a driver is that it properly formats a config file given the information that should appear within. This is a reflection of the system design which describes the site not just the config files in the site. There is a clean split between the description of the site data from the description of how a config file should look.

2.5. Pulling the Levels Together

Each level can be organized into one or more files. The separate files are `#include`'ed into a Sitefile for use as input to the Site system. The following sections illustrate how a Sitefile is changed to reflect different kinds of configuration activities in the site. Figure 2.6 is the initial state of the Sitefile which is altered in the following sections.

```
BrownWs bob {                                # Local Data level
    naddr =      42;
    display =    cg4;
    ...
}

class BrownWs : Sun3-60-S4 {                  # Local Types level
    memsize =    8 mb;
    ...
}

class Sun3-60-S4 {                            # Vendor Types level
    disksize =   140 mb;
    naddr =      ;
    display =    ;
    ...
}
```

Figure 2.6: A simple Sitefile

Adding An Object To The Site

Figure 2.7 illustrates how the Sitefile changes to reflect the addition of an object--in this case a computer--to the site. Sitefile variables are so called because they represent the parts of a site which vary from site to site. Each Sitefile variable represents a single real world site object such as a computer, network or printer. Adding such an object to the site involves adding exactly one variable entry to the Sitefile.

In this example we are adding the workstation mary. The bold entry in Figure 2.7 is the addition to the file. This workstation is the same type, and hence the same Sitefile "Type", as bob. The variable entry takes the same form as for bob.

The significant point of this illustration is that this type of change to the site requires a change only to the Local Data area of the Sitefile. This reflects our high-level goal of making the Sitefile relatively easy to edit as well as satisfying our data description goals of describing the site in the most "natural" manner.

<pre> BrownWs bob { naddr = 42; display = cg4; ... } class BrownWs : Sun3-60-S4 { memsize = 8 mb; ... } class Sun3-60-S4 { disksize = 140 mb; naddr = ; display = ; ... } </pre>	<pre> BrownWs mary { naddr = 137; display = cg4; ... } </pre>
--	---

Figure 2.7: Adding an object to the site

Factoring Common Information

If, for example, we are planning to add many workstations all of the same type we can factor out any commonality and place this common information in a lower level entry. This is illustrated in Figure 2.8. We have now factored out the commonality of bob and mary by moving this information to BrownWs.

```
BrownWs bob {
    naddr =      42;
    ...
}

BrownWs mary {
    naddr =      137;
    ...
}

class BrownWs : Sun3-60-S4 {
    memsize =    8 mb;
    display =    cg4;
    ...
}

class Sun3-60-S4 {
    disksize =   140 mb;
    naddr =      ;
    display =      ;
    ...
}
```

Figure 2.8: Adding an object to the site

Local Customization, Extensibility

A number of features permit local customization and extensibility of the site description. The Sitefile format has no built-in types or attributes and therefore no built-in notion of what anything in the file means. A Sitefile only organizes information. There is no predefined or required way to organize the site info. This permits the local site administration to describe their local modifications and customization **in the same manner** as any standard pieces of information.

In Figure 2.9 we present a simple example of this. The vendor's standard notion of their product offering doesn't include a notion of office location. It is significant to notice what changes as well as what *doesn't* change. Since the changes reflect how a Sun3-60-P4 is used at the local site the changes are isolated to the Local Types area and nothing changes in the Vendor Types area. In fact all Vendor Type Entries should be considered "read only" and ideally are produced at the factory along with the vendor's price book. Any changes a vendor's idea of what their product is should be made in the

```

BrownWs bob {
    naddr =      42;
    office =    575;
    ...
}

BrownWs mary {
    naddr = 137;
    office = 576;
    ...
}

class BrownWs : Sun3-60-S4 {
    memsize =    8 mb;
    display =    cg4;
    office =     ;
    filesys =    /cs NFS;
}

class Sun3-60-S4 {
    disksize =   140 mb;
    naddr =     ;
    display =    ;
    filesys =    / at sd0a;
    filesys =    /usr at sd0g;
    ...
}

```

Figure 2.9: Customization and Extensibility

Local Type area of the Sitefile. We have been using the Type BrownWs to isolate any local changes to a Sun 3/60 from the Type Sun3-60-S4.

We have also added a filesystem Attribute to BrownWs. This describes a local enhancement where all the site's workstations NFS-mount the directory "/cs".

More Local Customization and Extensibility

The example in Figure 2.10 illustrates the use some Sitefile features useful in specifying configuration policy for the site. In this case we are stating that our workstations are YP clients which implies that their /etc/hosts file contain minimal information required in the boot process. Our server machine is a YP server whose /etc/hosts file should contain hosts database information for the whole site. It is important to note that these changes to the description do not require any modification to the entries supplied by the vendor nor do they effect how one creates entries at the Local Data area. Adding a new workstation to the site is still the same procedure as above even if the site's name lookup policy were changed. This illustrates the power of the object oriented concept of type hierarchy in general and the utility of this concept in organizing site information.

```

BrownWs bob {
    naddr =      42;
    office =    575;
    ...
}

BrownWs mary {
    naddr =      137;
    office =    576;
    ...
}

BrownServer kelner {
    naddr =      55;
}

class BrownWs : Sun3-60-S4 YPClient {
    memsize =    8 mb;
    display =    cg4;
    office =      ;
    ...
}

class BrownServer : Sun4-280 YPServer {
    memsize =    32 mb;
}

class YPClient {
    /etc/hosts := StubHosts();
}

class YPServer {
    /etc/hosts := FullHosts();
}

class Sun3-60-S4 {
    disksize =   140 mb;
    naddr =      ;
    display =    ;
    ...
}

```

Figure 2.10: Customization and Extensibility

2.6. Object Oriented

The Sitefile demonstrates the power of object oriented data representation. Object oriented design is significant enough in the character of this work to warrant a special section on the topic. The following paragraphs summarize use of specific object oriented concepts.

Factoring Common Information

Factoring common Sitefile attributes and components into lower-level types allows almost any single piece of site configuration information to appear in one and only one place in the Sitefile.

Extensibility

Defining new classes based on old ones allows extensibility without modification of existing classes. This is important for maintaining commonality with a standard configuration. The new classes specific to a site precisely describe how that site differs from the default configuration.

Exceptions to Classes

Often times one is faced with classifying an object which is mostly of a certain type but differs in a few minor aspects. The attribute and component look-up mechanism permits an effective way to describe objects which are roughly of a certain type but which differ only in a couple of attributes or components. Since a Sitefile variable entry is consulted first it may contain attributes or components which over-ride a default value specified lower in the type hierarchy.

Multiple Inheritance

The Sitefile uses multiple inheritance to permit "mixing" of different types of objects. For example, a computer which plays the role of file server, print server, and network gateway can be represented by classes for each of these roles.

Multi Level

We have presented Sitefiles divided into 3 levels. The hierarchical typing system permits clean separation of these levels. This is useful for organizing information effectively, but, more importantly, also has the effect of separating complex and simple Sitefile entries. The significant result is that a novice may work at the highest level in terms comfortable to them with the information they manipulate expanding and flowing into the more complex machinery at the lower level.

2.7. Theory of the Site System

The Site system incorporates most of the ideas presented in this section. See Chapter Three for the implementation details of the Site system and details of features which are not based on ideas presented in this section.

A Sitefile represents a set of directed graphs. A graph is formed by variables and types representing nodes connected by edges pointing from base types to derived types and from types to variables. All non-terminal nodes are types, and each terminal node is a variable. (A non-terminal node has no out-pointing edges.) Each node has data in it containing the attribute and component entries of the corresponding variable or type entry. Types derived from more than one type have one inpointing edge for each base type.

See Figure 2.11 for an illustration of part of Figure 2.10 as a graph. The arrows point in the direction of data flow. This picture perhaps makes more apparent the scope of attributes and components implied by type inheritance. For example, the component `/etc/hosts := StubHosts();` "flows" to `BrownWs` and further to `bob` and `mary`.

The purpose of the graph is to model the flow of data from the objects to form information useful in producing a config files. The source of the data flow is the information about objects in the site, information about the site as a whole. The initial design tried to capture the flow of data all the way into specific syntactic pieces of each config file. It proved too complicated to cover the full diversity of file formatting required of the wide variety of config files. The current design, therefore, decouples the description of data from the description of how to format it. The former can be done with declarative data-description constructs such as hierarchical types, while the latter requires more general purpose programming constructs of sequences, conditionals, and loops. The present design gathers all of the information needed to produce the file and leaves the formatting of the file to a *driver* which takes this information as input. The drivers hide their implementation and are hidden from the structuring of the information in the graph. The bulk of this thesis deals with data description side of this scheme and generally leaves the machinery for formatting each file to systems which are already well suited for this such as the UNIX tools of `awk` and `sh` and the C language.

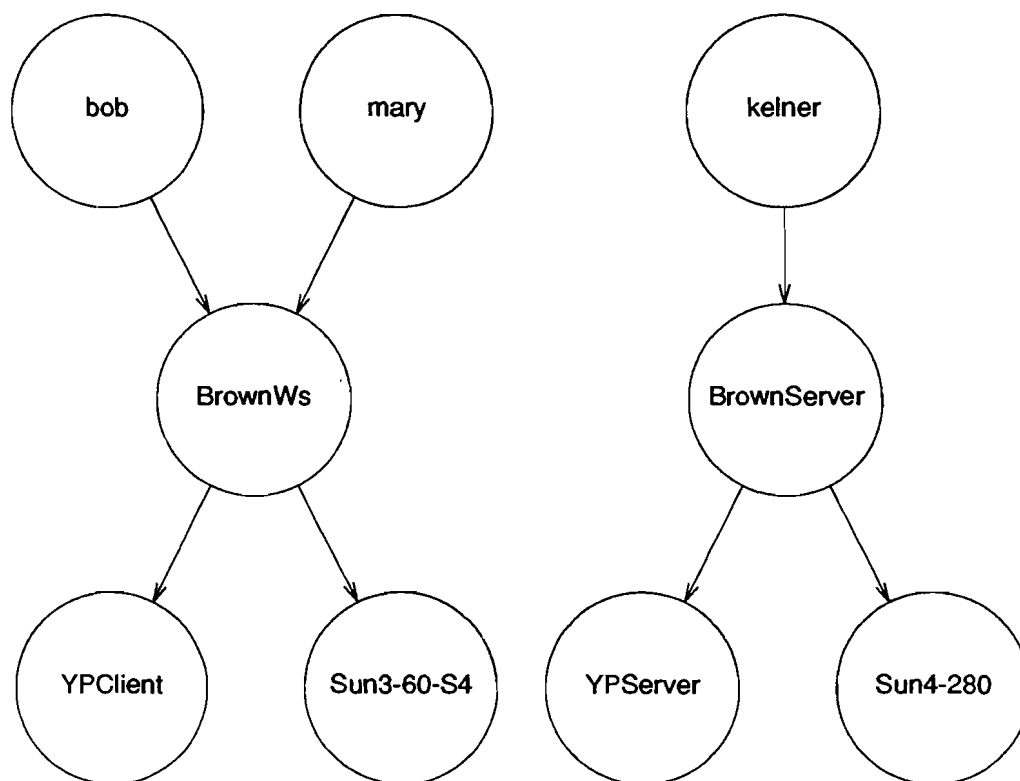


Figure 2.11: Graph of Sitefile

Data Flow Through the Graph

The following data flow "algorithms" describe how the graph is used. The overall idea is to traverse the graph and produce on output a stream of *hooks* --calls to driver programs. The traversal essentially "flattens" the hierarchy inherent in the directed graph and computes an *environment* of name/value information which each driver uses to produce a config file.

Data flows through the graph in two ways, each corresponds to a dependency between a config file and the information in it. In Chapter One we highlight the source of several individual pieces of information for a set of config files. There are config files which depend only on information about the computer on which the config file is located; and there are config files which depend on information about many other objects in the site.

For example, the kernel configuration file for a computer depends only on information about that computer such as the hardware devices connected to it, while the `/etc/printcap` file depends on information about which printers exist and information about each printer such as its print server.

The two data flow algorithms are presented in Figure 2.12. Each algorithm traverses a graph gathering up an *environment* as it visits each node. The environment consists of a set of triples where each triple consists of a variable name, field name, and field value. The variable name is included to group a set of attributes on a per-variable basis. The edges point in the direction of data flow from base types to derived types and from types to variables. Conflict resolution on attribute names and component names is designed such that a variable or type may specify its own value or driver for an

`per_variable_flatten(input: a graph)`

terminal_node:

- 1) do single_node_gather
- 2) for each component gathered
 - a) do per_type_flatten
 - b) return/emit call to driver with gathered environment

single_node_gather:

- 1) gather env from parent
- 2) gather env from current node
(env on current node overrides env from parent)

multi_parent_node_gather:

- 1) gather env left->right from each parent
(left overrides right on conflict)
- 2) consider the gathered env that of a "pseudo-parent"
- 3) do single_node_gather

`per_type_flatten(input: graph, type node, attribute names)`

- 1) generate list of downstream terminal nodes
(all variables of that type)
 - 2) do a per_variable_flatten of the graph
 - 3) for each variable grab any attributes of interest (omit components)
 - 4) return environment
-

Figure 2.12: Sitefile Graph Data Flow Algorithms

attribute or component, respectively.

Example Output of Graph Traversal

Figure 2.13 contains the output of traversing the graph in Figure 2.11 (which is the Sitefile in Figure 2.10). The intention is to feed this command stream to a UNIX shell whose command search path includes a directory which contains the programs StubHosts and FullHosts. These programs are the drivers whose implementation is external to this system and hence not specified. A typical driver is easily implemented with a simple awk or sh program.

```
StubHosts bob < "mary,naddr,137;bob,naddr,42;kelner,naddr,55;..." > etc.hosts.bob
StubHosts mary < "mary,naddr,137;bob,naddr,42;kelner,naddr,55;..." > etc.hosts.mary
FullHosts kelner < "mary,naddr,137;bob,naddr,42;kelner,naddr,55;..." > etc.hosts.kelner
```

Figure 2.13: Sitefile Graph Data Flow Algorithms

2.8. Future Work

The current implementation of Site requires one to relinquish control of configuration files to Site and a Sitefile. A facility for "reverse compiling" existing configuration files coupled with some heuristics for resolving conflicts between the information inherent in a configuration file and the information in a Sitefile would permit direct editing of configuration files under the control of site. A more important use for such a capability would allow implementation of a finer grain "incremental" change mechanism. That is, Site would be smart enough to generate only those files which need changing based on an incremental change to the Sitefile.

We do not address network security or authentication. The Sitefile "language" described in this work does not allow expression of the concepts of administrative domain and ownership of parts of a site. One could possibly extend the language to permit ownership of update permission to types, variables, components or attributes and extend the implementation of Site to use this information -- presumably along with a network authentication mechanism.

The most radical implementation would call for the outright removal of all configuration files. The functionality of attribute lookup currently used by drivers to scan their environment could instead be moved to the C library functions used to peruse the "/etc/blah" file. For example, the implementation of the `getfsent()` routine could be replaced by (network transparent) Sitefile calls. This already

```

class Ws {
    hostname =          ;
    ...

    "/usr" :=           Usr();
    "/dev" :=           Dev();
    "/usr/local" :=     UsrLocal();
    ...

    MountPoints();      /* fstab dirs */
    LpdSpoolDirs();      /* printcap dirs */
}

```

Figure 2.14: Future Sitefile entry

has precedence in the implementation of C library functions such as the `getpw*`() and `gethost*`() for use with YP.

The current implementation of Site uses Sitefile components to represent configuration files. This implementation could be extended to permit use of Sitefile components to describe specific directories which need to exist with certain mode, owner and group settings, as well as special--/dev--files, and finally for actual software "subsets". Rdist Distfiles already can handle the transport issues of most of these situations. A more comprehensive implementation of Site would allow processing of Sitefile entries such as the one in Figure 2.14.

CHAPTER 3

Other Attempts

3.1. Other Approaches

There are very few systems designed for configuring a site as a whole. This chapter briefly mentions systems related to managing a site's configuration and relates this work to other tools which inspired the design of Site.

Many ideas in Site stem from the experience of using rdist [3] and Distfiles to manage the configuration of a Computer Science Department site consisting of 100+ workstations and servers from several vendors. A discussion of some of the specific problems is included in this chapter.

3.2. rdist

The strategy in using rdist to manage the configuration of a large site was to gather up all configuration files for all machines into a directory on a server. A very complex Distfile (~1000 lines) described how these files were to be installed on various machines. The most difficult problem in using rdist for this task was in trying to impose a classification scheme for the various types of machines in the site. Beyond the basic mechanism of providing the actual transport of files to remote machines there are few facilities for actually producing the configuration files let alone the managing of the information needed to produce the files.

Rdist works fine as a file transport mechanism and for specifying which files and directories should be distributed to which machines. The most important limitation for the purpose of maintaining configuration files is its lack of general ability to organize information.

More specifically rdist has no method for declaring types or for classifying information, and it's difficult to factor out common information and express "minor" exceptions to a rule.

Rdist example

The following example will illustrate some problems in using a Distfile. A goal in maintaining the Distfile is to mention each hostname and each unique file once in the Distfile. This is accomplished largely by using macro definitions as much as possible in the rules; these macros essentially define classes of hosts and files. This is all straight forward in the case where the intent is to distribute exactly the same files to a set of hosts. The Distfile in Figure 3.1 distributes all of the same files to four hosts.

```
FILES=( ...{all files}... )
HOSTS=(a b c d)

${FILES} -> ${HOSTS}
install;
```

Figure 3.1: A Distfile

However, any single exception forces the split of a HOST "type" in to two. For example, all hosts are to be configured identically except for host d, which is to have a different /etc/passwd than the rest. The fact that a host needs a "special" /etc/passwd effectively introduces a new class of host. In the Distfile this is described by the new macro HOSTS_SPECIAL. Figure 3.2 shows the resulting Distfile.

```
CONF=/etc/conf
ALL_FILES=( ...{all files except /etc/passwd}... )

HOSTS_NORMAL=(a b c)
HOSTS_SPECIAL=(d)
HOSTS_ALL=(${HOSTS_NORMAL} ${HOSTS_SPECIAL})

${CONF}/passwd.special -> ${HOSTS_SPECIAL}
install /etc/passwd;

${CONF}/passwd.normal -> ${HOSTS_NORMAL}
install /etc/passwd;

${ALL_FILES} -> ${HOSTS_ALL}
install;
```

Figure 3.2: More Complex Distfile

Things get considerably more complex each time a new exception is introduced. If, for example, the host b needs a special `/etc/fstab` the Distfile would look as in Figure 3.3. There are now three classes of hosts. Clearly this could get out of hand quickly in a real site.

```
HOSTS_NORMAL=(a c)
HOSTS_SPECIAL=(d)
HOSTS_SPECIAL2=(b)
HOSTS_ALL=(${HOSTS_NORMAL} ${HOSTS_SPECIAL} ${HOSTS_SPECIAL2})
ALL_FILES=( ... {all files except /etc/passwd AND /etc/fstab } ... )

${CONF}/passwd.special -> ${HOSTS_SPECIAL}
install /etc/passwd;

${CONF}/passwd.normal -> ${HOSTS_NORMAL} ${HOSTS_SPECIAL2}
install /etc/passwd;

${CONF}/fstab.special2 -> ${HOSTS_SPECIAL2}
install /etc/fstab;

${CONF}/fstab.normal -> ${HOSTS_NORMAL} ${HOSTS_SPECIAL}
install /etc/fstab;

${ALL_FILES} -> ${HOSTS_ALL}
install;
```

Figure 3.3: Even More Complex Distfile

Fixing rdist

Initial designs for a site configuration tool began with attempts to fix rdist to better deal with problems such as the ones described here. One such improvement involved a richer set of operations for assembling files from templates and from smaller pieces of files. We discovered what we really wanted to do was describe the information in the files and to automate production of the contents of the files. This is a separate issue from how to transport a file from one machine to another in a network. We also discovered that our use of macro definitions for classifying hosts was very similar to an object oriented type hierarchy. We left rdist alone and moved on to work out a scheme for organizing configuration *information* along a classification of machines in the site. The key was to separate configuration information from the configuration files.

3.3. Make

Make [2] is generally used as a software engineering tool to describe dependencies between software components. Make was studied for its declarative rule-based style and was indirectly the source of the idea organizing site objects and parts into a multi-way tree or directed graph. That is, each config file to be *made* was put in a make rule which had dependencies on other rules which had to fire first to assemble sub-parts of the config file. Again, this did not lead to much success as the complexity of piecing together files by using things that could be called from a Makefile rule -- such as awk, sed, and cat -- became more cumbersome than describing a classification of the site.

A proposed extension to make added variables to each make rule. A variable set in a dependent rule stays active in the current rule unless the current rule overrides the value. The rule name and dependent rule names evolved into Site class or type names; and the variables and values evolved into Site attributes.

3.4. Interactive Programs

This section mentions some interactive utilities offered by various vendors familiar to the author. Sun's suninstall and Encore's devconfig, configure, and partition are examples of such programs.

Computer vendors more commonly approach assisting a systems administrator by providing "user friendly" utilities which front-end the preparation of various configuration files by presenting the administrator with an "easy to use" interactive program. Interactive programs either produce configuration files directly or preserve input in an internal (tool dependent) representation form.

There are several problems with this approach. While this may simplify initial setup, it doesn't provide much assistance for further on-going maintenance which largely consists of incremental changes. Another affect of this information loss is to prevent recording of what input was provided. Any interaction method should be optional and should operate only and completely on a common representation.

If the representation form is simple enough one can make direct use of tools specifically designed to provide management of incremental change such as RCS and SCCS.

Interactive tools often times require the administrator to fill out (paper!) work sheets before running the utility anyways!

3.5. config

The BSD UNIX kernel is created and configured by the *config* program [4]. This tool deserves mention here as some of the ideas used in Site were inspired by *config*.

The foremost idea is of automating the production of the UNIX kernel through the use of a "single, easy to understand, file". From this standpoint Site is merely an extension of *config* to describe not just the system kernel image, but the entire site. In fact, the syntax of Site attributes is a direct descendent of the device specifications found in a *config* file.

3.6. YP

The Sun Yellow Pages system [1] is a network-distributed lookup service. Standard UNIX database files such as */etc/hosts*, */etc/passwd*, */etc/group*, and */usr/lib/aliases* are replaced by corresponding YP "maps" while retaining the same library function interface used to access the files. While this system does address some problems in large systems administration it does not directly address site

configuration. In fact, YP itself must be configured on a per-host basis!

3.7. Summary

Little work has been done to solve the problems of site administration. Interactive tools certainly have appeal for ease of use, but this issue is really independent of the more important problem of accurately and precisely describing site information. For the work described in this thesis the ideas behind make, rdist and config are the most important.

CHAPTER 4

Implementation and Reference

4.1. Site 1.0 Implementation

This chapter describes "Site 1.0" which is the initial version of the Site system. Site 1.0 is based on the ideas presented in Chapter One. The information in this Chapter is restricted to Site 1.0 implemented for UNIX. (UNIX features described here are generally available on any version of UNIX). For information on using Site 1.0 refer to Appendix A.

4.2. Site 1.0 Modes of Operation

Site 1.0 consists of the command *site* which may be invoked in one of three modes.

In *interactive* mode *site* reads a Sitefile and produces a directory of config files along with a Distfile for distributing the files to remote machines.

In *client* mode *site* connects to a *site server*. A *site* command running in this mode is called a *site client*. A *site client* sends commands to a *site server*, and uses the information gathered from these commands to produce config files on the local host.

In *server* mode *site* reads a Sitefile and executes commands sent to its TCP socket. A *site* command running in this mode is called a *site server*. A *site server* accepts and executes commands and returns the response to the remote IPC port.

Each of the three modes is implemented in terms of modules which implement lower level features of the Site system including: Sitefile parsing, access to Sitefile variables, types, attributes, and components; access to config file drivers; low level abstract (C language) types used by all of these modules.

4.2.1. Interactive Mode

See Figure 4.1 for a pseudo-code representation of interactive mode. The function `readsitefile()` reads in the Sitefile and stores its contents in internal data structures. After the Sitefile has been completely parsed this mode calls `vareval()` which *evaluates* each Sitefile variable. Evaluating a Sitefile variable involves evaluating each the of the variable's components; and evaluating a component results in calling a *driver* to produce a config file. See the Sitefile Parsing section for more information on `readsitefile()` and the Variables section for information about `vareval()`.

```
interactive()
{
    readsitefile();
    for (each Sitefile variable)
        vareval();
}
```

Figure 4.1: Implementation of Interactive Mode

4.2.2. Client/Server Mode

In this section we summarize the implementation of *client* and *server* modes. See Figure 4.2 for the pseudo-code representation of *server* mode. As with *interactive* mode a *site server* begins by reading a Sitefile into an internal representation. The `servermode()` function implements the standard structure of a Berkeley UNIX network server. For each connection this function spawns a child process running the `server()` function which reads commands from the socket and writes the response of each command on the socket. Before describing the *site client* we present a summary of the "protocol" interface to the *site server*.

```

servermode()
{
    readsitefile();
    init a socket to listen on:
        socket()
        bind()
        listen()

    for(EVER)
        accept()
        fork()
            if ( child )
                server()

server()
{
    while ( getline from port )
        parse line into command and arguments
        execute command
}

```

Figure 4.2: Implementation of Server Mode

Site Protocol

This section describes each of the commands or queries accepted by the *site server*. All input to the server is in ASCII text as is all output. The output is returned on the same connection as input—the client process sends queries and reads the response to each on the same file descriptor. Most output consists of multiple lines. The first line echoes back a summary of the command. The next line indicates the number of following output lines. Each of the following output lines is typically a single string.

varlist The first line of output consists of the number of variables, each subsequent line consists of a variable name. The order of names returned has no meaning, need not be the same order in which they appear in the Sitefile and is not guaranteed to be consistent for each invocation of this command.

typelist var This returns the names of all types which *var* is based on. The first line of output is the number of lines to follow. Each following line is the name of a type.

varsoftype *type*

This returns the names of all variables based on *type*. The first line of output is the number of lines of output to follow. Each following line is the name of a variable. The final line of output consists of a single '.'.

attrval *var attr*

This returns the value of the single-valued attribute *attr* in the context of the variable *var*.

The first line of output is the number of words in the value of the attribute.

attrval2 *var attr val*

This returns the value of the specified attribute whose first value word is *val*.

attrvaln *var attr n*

This returns the value of the *n*-th occurrence of the specified attribute in the specified variable.

attrcount *var attr*

This returns the number of times the variable *var* specifies a value for the attribute *attr*.

attrlist *var* This returns a list of all attribute names found in this variable.

compcount *var*

This returns the number of components in the specified variable.

complist *var* This returns a list of the components of the variable *var*. Each line of output consists of two words: the first word is the target name (configuration file name), and the second word is the name of the driver.

compargs *var targ driv*

This returns the arguments to the driver which is uniquely specified by the variable name *var*, target name *targ* and driver name *driv*.

Site Client

See Figure 4.3 for the pseudo-code of a *site client*. The *site client* communicates with the server to get the information needed to create the configuration files for the local host--the host the *site client* is run on. To do so involves evaluating the variable associated with the the client host. The *site client* is implemented in terms of the Sitefile Access Library which in turn uses the Site Protocol to communicate with the *site server*.

The *site client* calls the function *SITEvareval()* which is a function in the Sitefile Access Library. This function gathers the names of all components in the variable representing the local host and calls the driver for each corresponding config file. A driver uses the *SITEattrval()* function to gather values of attributes needed to produce the config file. More about the *SITE*()* library appears in a later section.

```
clientnode()
{
    clientinit(serverhost)
    clienteval()
}

clientinit(serverhost)
{
    connect to server:
        socket()
        connect()
}

clienteval()
{
    SITEvareval(thishost)
}
```

4.3

4.3. Site 1.0 Low Level Modules

The following sections describe the interfaces to low level modules internal to Site 1.0. The most important modules are those which implement the interpretation of the Sitefile types and

attribute-value lookup; see the Types and Attributes sections respectively.

4.3.1. Site **LISTs**

The functions in this module implement the operations on the **LIST** type. A Sitefile is represented internally in terms of **LISTs**. See figure 4.4 for a summary of **LIST** functions used to construct or write to **LISTs**.

```
#include "list.h"

int listinit(LISTP listp, int max, char *name, int type);
/*
 * initialize a new list to have a maximum of "max" elements
 * give it a name and a type (see LIST_* above)
 */

int listset(LISTP listp, int ix, caddr_t val);
/*
 * set the "ix"th element of the list to "val"
 */

LISTP listfromchain(LISTP listp, CHAINP chp);
/*
 * put the elements in the linked list "chp" into
 * the list "listp". this calls "listinit()".
 */

int listadd(LISTP listp, caddr_t obj);
/*
 * Append "obj" to "listp"
 */

int listcat(LISTP listp, LISTP catlistp);
/*
 * concatenate the elements in "catlistp" to the end of "listp"
 */

listfromfio(LISTP listp, FIOF fiof);
/*
 * Produce a list of STR from lines found in the stream "fiof"
 * The first line in "fiof" must be a number which will be taken
 * as the number of lines of input to put in the list
 */

listfromfiopairs(LISTP listp, FIOF fiof);
/*
 * Produce a list of PAIR from lines found in the stream "fiof"
 * The first line in "fiof" must be a number which will be taken
 * as the number of lines of input to put in the list. Each
 * following line must have 2 space separated strings.
 */
```

Figure 4.4: LIST Construction Functions

Figure 4.5 summarizes the routines used to read LISTS. The example in Figure 4.7 describes how some of these routines are used.

```
int listiterinit(LISTP listp, ITERP iterp);
    /*
     * Initialize the list iterator "iterp" for "listp"
     */

caddr_t listiter(LISTP listp, ITERP iterp);
    /*
     * Get the next element in the iteration
     * ("listiterinit()" should be called before using this)
     */

int listlen(LISTP listp);
    /*
     * return the current number of elements in "listp"
     */

int listcap(LISTP listp);
    /*
     * return the capacity "listp". This is the "max" value
     * supplied to 'listinit()'.
     */

int listtype(LISTP listp);
    /*
     * return the type (LIST_*) of the list
     */

caddr_t listmem(LISTP listp, int nx);
    /*
     * Return the "nx"th object in "listp"
     */

caddr_t listfirst(LISTP listp);
    /*
     * Return the first object in "listp"
     */
```

Figure 4.5: LIST Reading Functions

4.3.2. Variables - var*()

This module implements access to Sitefile variable entries. The interface consists of the functions described in Figure 4.6. Internal to Site a Sitefile variable is represented by a C variable of type VAR. A VAR describes the variable entry from the Sitefile and describes the Sitefile variable name, the Sitefile type name and the body of the variable entry.

```
LISTP varlist()
/*
 * Return a pointer to the LIST of VARPs representing
 * all variables in the Sitefile.
 */

char *vamame(VARP varp);
/*
 * Return the name of the given variable.
 */

char *vartype(char *vname);
/*
 * Return the name of the type of the given variable.
 */

LISTP varbody(char *vname);
/*
 * Return a pointer to the LIST representing the
 * body of the variable entry of the specified variable.
 */

vareval(char *vname, char *dir)
/*
 * evaluate the variable 'vname', put the config files
 * in the directory 'dir', add an entry to the Distfile
 * in 'dir'.
 */
```

Figure 4.6: Sitefile Variable Functions

Figure 4.7 gives an example of how the Variables and LISTs modules are used internal to Site. The variable **lter** of type ITER holds the state of the iteration through the LIST pointed to by vl. Each call to listiter() returns a pointer to the next element in this list; after the end of the list listinter() returns NULL.

```

sitefileeval(dir)
char *dir;
{
    /*
     * evaluating a sitefile means to evaluate each variable
     */

    LISTP vl;
    ITER iter;
    caddr_t varp;

    distinit(dir);

    vl = varlist();
    listiterinit(vl, &iter);
    while ( varp=(caddr_t)listiter(vl, &iter) )
        vareval(varname(varp), dir);

    distfini();
}

```

Figure 4.7: Example use of LISTS and Variables

4.3.3. Types - type*()

The Type module provides access to type entries and implements the "flattening" of the type hierarchy. The function `typelist()` is very significant in the implementation of Site 1.0 in that it exclusively embodies and hides the interpretation of type hierarchies in the Sitefile. Given a typename it returns a list of typenames in the order of look-up precedence. There is one important implementation assumption made in doing so. This assumption is that the meaning of the precedence implicit in a type hierarchy described in the Sitefile--essentially as a multi-way tree--can be reduced to linear precedence list. For example, the `attrval()` function does not operate on the type hierarchy as tree, but as a list of types the ordering of which is used to resolve conflicts on attribute names. That is, when searching for the value of an attribute in the given variable it returns the value found in the first type in the list returned by `typelist()`.

```

LISTP typebody(char *typename)
    /*
     * Returns a pointer to the LIST representing the body
     * of the type entry for the type named 'typename'.
     */

typelist(LIST &list, char *typename)
    /*
     * Produces a list of the typenames representing the "flattened"
     * type hierarchy on which the type 'typename' is based.
     */

```

Figure 4.8: Sitefile Type Interface

Linear Interpretation of the Type Hierarchy

The function `typelist()` returns a linear ordering of the type hierarchy as follows. The first type in the list is always the base type -- the typename given to `typelist()`. This is immediately followed by the super type names in the order found in the type entry for the type `typename` -- in the Sitefile type entry these are the names following the `':'`. The rest of the list is the result of a depth-first search down each super type. For example, given the (skeletal) type entries in Figure 4.9 `typelist()` would return the list:

CsSun-3Disked Sun-3/60Disked CsClient CsMachine Sun-3/60Base SunBase Machine.

```

class CsSun-3Disked : Sun-3/60Disked CsClient CsMachine { ... }

class Sun-3/60Disked : Sun-3/60Base { ... }

class Sun-3/60Base : SunBase { ... }

class SunBase { ... }

class CsClient { ... }

class CsMachine : Machine { ... }

```

Figure 4.9: Skeletal Type Hierarchy

4.3.4. File I/O - fio*()

Routines for file I/O are documented in Appendix A.

4.3.5. Bodies - body*()

The Body module implements access to the body of a variable or type entry. These routines are used by the Attribute and Component modules. Figure 4.10 summarizes the Body interface.

```

int bodyn(LISTP bodyp, int which);
/*
 * Return the number of attributes/components (which == ATTR/COMP)
 * in the body represented by "bodyp". Use "varbody()" or
 * "typebody()" to get the body of a variable or type entry.
 */

int bodylist(LISTP listp, LISTP bodyp, int which);
/*
 * Put in "listp" all attrs/comps (which == ATTR/COMP)
 * found in the body "bodyp". Use "varbody()" or
 * "typebody()" to get the body of a variable or type entry.
 */

```

Figure 4.10: Sitefile Body Interface

4.3.6. Attributes - attr*()

The Attributes module implements attribute lookup in variables. See Figure 4.12 for a summary of the interface to Attributes. The attribute-value look-up functions in this module use the `typelist()` function described in the Types Section for computing the basis for conflict resolution.

There are two different ways to determine the value of an attribute for a given variable. The first way is to use `attrval()` for a *single-valued* attribute; and the second way is to use `attrvaln()`, `attrcount()`, and `attrval2()` for a *multi-valued* attribute. The Sitefile does not specify which attributes are single valued and which are multi valued. The interpretation of an attribute as single valued or multi valued is left to the routines using the `attr*()` functions.

Single-Valued Attribute Lookup

We present single-valued attribute look-up using the example of a variable which represents a networked computer and the hostname attribute. A routine which operates on variables which represent networked computers may need the value of the hostname attribute. Since a computer is known by a single hostname this routine will use the `attrval()` function to carry out a *single-valued* attribute lookup. The search for the value of a single valued attribute begins in the variable entry. If no value for the attribute is found in the variable entry the search continues into the type "hierarchy" in the order of the type entries whose names are returned by `typelist()`. If a search of a given variable or type entry reveals that there is more than one attribute of the given name in that entry an error is returned.

Multi-Valued Attribute Lookup

A routine which needs to find the disks connected to a given computer may use multi-valued attribute lookup on the disk attribute. Such a routine would use `attrcount()` to determine the number of disk attributes and would use `attrvaln()` to get the value of each instance of this attribute.

The function `attrval2()` is useful on an attribute which has a multiple word value. See Figure 4.11 for example attribute entries with multiple word values. Using `attrval2(..., "filesystem", "/")` would return the first attribute.

```
filesystem = / at sd0a;  
filesystem = /usr at sd0g;
```

Figure 4.11: Multiple Word Attributes

```
int attrtotal(char *varname);
    /*
     * Returns total number attributes
     */

LISTP attrval(char *varname, char *attrname);
    /*
     * Returns value of single valued attribute
     */

LISTP attrval2(char *varname, char *attrname, char *val);
    /*
     * Returns attribute whose first val word is "val"
     */

LISTP attrvaln(char *varname, char *attrname, int n);
    /*
     * Returns the 'n'th attribute named "attrname"
     * Attributes are in top->bottom order first by variable body,
     * then by type body in order of type level and left->right in
     * each level.
     */

char *attrname(LISTP attrp);
    /*
     * Returns name of attribute
     */

int attrcount(char *varname, char *attrname);
    /*
     * Returns number of attributes named "attrname"
     */
```

Figure 4.12: Sitefile Attribute Access Functions

4.3.7. Components - comp*()

```
char *comptarg(LISTP comp);
/*
 * Returns target name of the component
 */

char *compdrv(LISTP comp);
/*
 * Returns driver name of the component
 */

int comptotal(char* varname);
/*
 * Returns the total number of componets of the specified variable
 */

int complist(LISTP compl (RETURN), char *varname);
/*
 * Returns a list of all components of the specified variable
 * "compl" should be a pointer to a LIST
 */

LISTP compargs(char* varname, char *targ, char *cname);
/*
 * Returns the component of the specified variable with
 * the specified target and name
 */

int compeval(LISTP comp, char *varname);
/*
 * evaluate the specified component
 * "varname" identifies the associated variable
 */
```

Figure 4.13: Sitefile Component Interface

4.3.8. Sitefile Parsing - readsitefile()

This module implements the *readsitefile()* function and consists of the yacc grammar and lex file which implement the lexical analysis and parsing of a Sitefile. The yacc code constructs a LIST of variables and a LIST of types of the variable and type entries found in the Sitefile.

4.3.9. SITE Access Library

This module consists of a C language interface to the Site Protocol. This set of function calls is documented in the Sitefile Access Library section in the Site User's Guide in Appendix A.

4.3.10. Drivers - driv*()

The whole purpose of describing site information in a Sitefile is to automate the production of configuration files. Drivers are part of the implementation of the Site program.

```
EtcHosts(target, arglist, thisvar, ofile)
char *target;
LISTP arglist;
char *thisvar;
char *ofile;
{
    FIO fio;
    LIST vl;
    char *varn;
    ITER iter;

    LIST hostname;      /*
    LIST hostnumber;    * attributes
    LIST ipsubnet;      */

    SITEfioopen(&fio, ofile);
    SITEfioprintf(&fio, "# /etc/hosts for %s0, thisvar);
    SITEfioprintf(&fio, "# generated from %s0, ofile);

    SITEvarnamelist(&vl, thisvar);

    SITElistiterinit(&vl, &iter);
    while ( varn=(char*)SITElistiter(&vl, &iter) ) {
        SITEattrval(&hostname, varn, "hostname");
        SITEattrval(&hostnumber, varn, "hostnumber");
        SITEattrval(&ipsubnet, varn, "ipsubnet");

        if ( SITElistlen(&hostname) == 0 )
            continue;

        SITEfioprintf(&fio, "%s %s.%s0,
                        SITElistfirst(&hostname),
                        SITElistfirst(&ipsubnet),
                        SITElistfirst(&hostnumber));
    }
    SITEfioclose(&fio);
}
```

Figure 4.14: Example driver

A driver is a C function which uses a set of Sitefile access routines to look up attribute values in variables as the source of information needed to produce the corresponding configuration file. For example, the VendorFStab driver looks for disk attributes of the variable for information needed to produce an /etc/fstab file. Additionally, the VendorKernel accesses the same information for its needs in producing a /vmunix file. See Figure 2.11 for an example driver for a full /etc/hosts file.

The Hosts driver in Figure 2.11 produces a file (named in `ofile`) in the format of an /etc/hosts file. `SITEvnamelist()` returns a list of names of all variables defined in the Sitefile. The while loop iterates through this list looking up the values of the hostname, hostnumber, and ipsubnet attributes for each variable. Any variable with a hostname attribute will yield an entry in the file.

4.4. Sitefile Grammar

The following is the formal definition of Sitefile format.

file:
 fileforms

fileforms:
 empty
 files fileform

fileform:
 variable
 type

variable:
 typename varname body

type:
 class typename body
 class typename : namelist body

body:
 { bodyform }

bodyform:
 empty
 bodyform form

form:
 aform ;
 cform ;

aform:
 name = ;
 name = namelist ;

cform:
 name := ;
 name := name (arglist) ;
 name (arglist) ;

arglist:
 empty
 arg
 arglist , arg

arg:
 name = name

namelist:
 empty
 namelist name

Figure 3.13: Sitefile grammar

APPENDIX A

User's Manual

5.1. Overview of Using Site 1.0

Site 1.0 consists of a single command called *site*. This command can be used in one of three modes:

- interactive mode,
- server mode,
- and client mode.

In *interactive* mode the administrator prepares a Sitefile which *site* takes as input. The output is a directory containing properly formatted config files along with a Distfile for distributing them to the machines on which they belong. *Interactive* mode represents a "push" or "one to many" model of maintaining site configuration.

In *server* mode the administrator uses the same Sitefile as above and runs *site* with an option which tells it to run as a "Sitefile server" for Sitefile access requests from remote *client* mode *site* commands. In *client* mode *site* connects to the server and gathers information needed to produce configuration files only for the local host. The *site client-server* mode represents a "pull" or "many from one" model.

The first section presents some hints on how to organize a Sitefile. The next section describes the Sitefile programmatic interface used in writing config file drivers. The final section describes how to run the *site* command.

5.2. How To Prepare a Sitefile

Refer to Chapter One for a detailed presentation of the syntax and semantics of a Sitefile. In this section we restrict our comments to helpful hints in organizing and using the Sitefile.

The following summarizes the basic content of the Sitefile. The Sitefile has one variable for each object in the site. Each variable is defined to be of a single specific type. All types must be defined in the Sitefile. There are no built-in types, variables, attributes or components.

It is suggested that a Sitefile be organized into 3 parts and that these 3 parts be placed in different files. For the remainder of this discussion we will assume that this is the case and will refer to the 3 parts of the Sitefile as the Vendor Types file, the Local Types file and the Local Data file. (Site 1.0 uses the UNIX C preprocessor to implement `#include` and `#define` directives.) Site 1.0 assigns no meaning to what's found in a particular file or even the order of Sitefile entries as resulting from the output of `#include`'ing each of the files together into one.

The Vendor Types file contains vendor specific classification information. (Ideally the Sitefile entries of this variety would be supplied by the vendor and would be part of the distributed systems software). This part should be considered "read only" and should only be changed to reflect changes in a vendor's product offerings. This part should contain enough default information to enable the production of valid config files even if there are no Sitefile entries in the Local Types part. The resulting configuration thus represents the vendor's default uncustomized "vanilla" configuration for their systems.

The Local Types files of the Sitefile contain site specific classification information. Most type entries in the Local Types file are based on types found in the Vendor Types file. Any hardware or software systems developed locally at the site probably will not be based on any Vendor Type entries.

The Local Data file contains the variable entries representing the inventory of all objects in the site. There is one variable entry here for each individual workstation, server, printer, network, etc, in the site. The most common operation on this file will be to make new entries or get rid of entries as objects are added or removed from the site. In general the first Local Data entry of a particular type of object will be created by a person with systems programming expertise usually when the corresponding Local Types entry is created for this object as it's tailored for use in the site. Later, when a new object of this type is added to the site a less experienced administrator edits this file by copying this first entry and then altering it to reflect information which makes this object different than the first one

– variable name, hostname, network address, etc.

5.3. How To Write a Driver

This section describes the programmatic interface used to access Sitefile data from a configuration file driver. There is a library of routines available for retrieving information from the Sitefile about attributes, components, types and variables. Also available are routines for manipulating LISTs and for file I/O.

Drivers generally come in two flavors which reflect the source of the data needed to complete the corresponding config file. The first flavor of driver produces a config file based on attribute values of only the associated variable. The other flavor produces a file based on attribute values of some if not all other variables. A driver may look for attribute values in all variables derived from a specified type or it may look through the attributes of all variables using its own heuristics to determine which variables are of interest. A driver may be a mix of both or may produce "constant" information not based on the attribute values of any variables.

5.3.1. Driver Calling Sequence

When a component is evaluated Site 1.0 makes a C function call to the component's driver. All drivers are called as follows:

```
Driver(target, arglist, thisvar, ofile)
char *target;
LISTP arglist;
char *thisvar;
char *ofile;
```

target	Path name of the config file.
arglist	LIST of PAIRs of arguments to thisvar's instance of the component.
thisvar	The variable name of the associated variable.
ofile	The path name of the output of this driver.

A driver is called with the component's target name which is the string on the left hand side of the assignment operator (:=) from the Sitefile component. The arglist is the list of PAIRs of left hand

side and right hand side names found between the “()” of the component. The driver is also passed the name (**thisvar**) of the variable associated with this component. Finally, the driver is passed the name of the file to which it should direct its output (**ofile**).

See Figure 3.14 Chapter 3 for the C code of an example driver.

5.3.2. Sitefile Data Access Routines

This section documents the routines available for use from the drivers to access data in the Sitefile. Internal interfaces to variables, types, attributes and components is documented in the thesis Part 3: Implementation Description.

Names of all Sitefile Variables

```
SITEvarnamelist(listp)
LISTP listp; /* RETURN */
```

This This fills in the LIST with the names of all the variables found in the Sitefile

Names of All Variables of a Given Type

```
SITEvarsoftype(listp, type)
LISTP listp; /* RETURN */
char *type;
```

listp Pointer to a LIST.

type Name of type.

This fills in a LIST of the names (STR) of all variables of the specified type. LISTP must be a pointer to memory allocated as a LIST.

Evaluate Variable

```
SITEvareval(vname, dir)
char *vname;
char *dir;
```

vname Variable name.

dir Directory to place config files.

This find all components of the named variable and calls SITEcompeval for each.

Evaluate Component

```
SITEcompeval(targ, driv, vname, dir)
char *targ;
char *driv;
char *vname;
char *dir;
```

targ Target config file name.

driv Name of driver.

vname Variable name.

dir Directory to place config file.

This calls the named driver -- an internal table maps the name to a function. The completed config file is placed in a file in dir with a name of the drivers choosing. The targ name is entered in a Distfile in dir in a rule which specifies how to copy the completed config file to this name on the host represented by the variable.

Value of a Single-Valued Attribute

```
LISTP
SITEattrval(varname, attrname)
char *varname;
char *attrname;
```

varname Name of the variable.

attrname Name of the attribute.

This returns a pointer to a LIST representing the first single-valued attribute found in the hierarchy associated with the variable and its type. It returns NULL if there are multiple instances of the attribute or if there is no such attribute in the variable. uh "Constrained Value of a Single-Valued

Attribute"

```
SITEattrval2(listp, varn, aname, val0)
LISTP listp; /* RETURN LIST_STR */
char *varn;
char *aname;
char *val0;
```

listp Pointer to LIST for return value.

varn Variable name.

aname Attribute name.

val0 First word of attribute's value.

This does the same thing as SITEattrval with the additional constraint that the first word of the value of the attribute is val0.

Count of Attributes in Variable

```
SITEattrcount(vname, aname)
char *vname;
char *aname;
```

vname Variable name.

aname Attribute name.

Returns the number of attributes named aname in the specified variable. Useful in multi-value attribute lookups.

Multi-Value Attribute Lookup

```
LISTP
SITEattrvaln(varname, attrname, n)
char *varname;
char *attrname;
int n;
```

varname Variable name.

attrname Attribute name.

n Which attribute

Returns the *n*-th attribute named *attrname* in the specified variable. Attributes are in top to bottom order first by variable entry, then by type entry in order of type level and left to right in each type level.

5.3.3. List Access Routines

For information about LIST routines see the Site LISTs section in Chapter 3.

5.3.4. File I/O Routines

The following routines front-end the UNIX stdio routines. An FIO records the file name and stdio FILE pointer.

```
SITEfioopen(fiop, ofile)
FIOF fiop;
char *ofile;
```

```
SITEfioprintf(fiop, format, arg,...)
FIOF fiop;
char *format;
char *arg;
```

```
SITEfioclose(fiop)
FIOF fiop;
```

fiop Pointer to a FIO.

format Pointer to a **printf()** format specification string.

ofile Name of output file.

5.3.5. Example: Use of Sitefile, list and file I/O Routines

The following code fragment demonstrates the use of the Sitefile and list access and file I/O routines. This code fragment creates the file `/tmp/ofile` and writes into it a line for each variable containing the value of the name attribute of each variable.

```
{  
    FIO fio;  
  
    LIST list;  
    ITER iter;  
    STR varname;  
  
    SITEfioopen(&fio, "/tmp/ofile");  
  
    SITEvarnamelist(&list);  
  
    SITElistiterinit(&list, &iter);  
    while ( varname=(STR)SITElistiter(&list, &iter) ) {  
        LIST name;  
        SITEattrval(&name, varname, "name");  
        SITEfioprintf(&fio, "%s: ", varname);  
        SITEfioprintf(&fio, " name=%s0, SITElistfirst(&name));  
    }  
}
```

5.4. How To Run Site

NAME

site - configure a site using a Sitefile

SYNOPSIS

site [-f Sitefile]
site -S
site -C sitehost

DESCRIPTION

Site is a configuration file compiler. A Sitefile describes the information needed to produce config files for computers in the site.

INTERACTIVE MODE

In the first mode *site* reads in the specified *Sitefile* or if one is not specified it uses the file *Sitefile* in the current directory. The output of *site* is a directory called *.lconf* which will be populated with config files for all systems in the site -- as specified in the *Sitefile*. A *Distfile* is also created in this directory which can be used by *rdist(1)* to distribute these files to the computers in the site.

CLIENT-SERVER MODE

When in invoked with the -S option *site* reads in the *Sitefile* and listens for commands on a network port. When in invoked with the -C option *site* connects to the site server on the specified host to evaluate the variable associated with the local host.

FILES Sitefile
conf/Distfile

SEE ALSO

rdist(1)

Site - A Language and System For Configuring Many Computers as One Computing Site