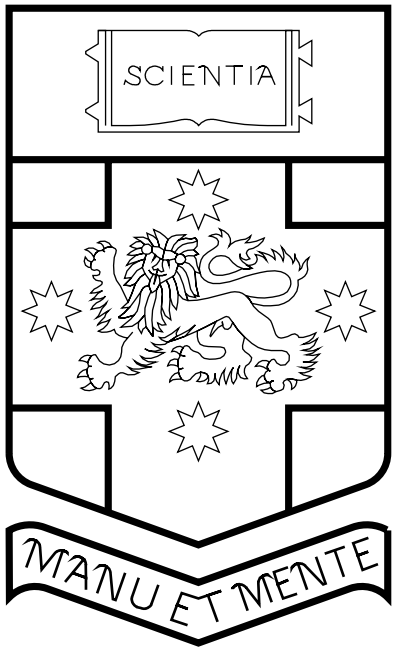
University of New South Wales

School of Computer Science and Engineering



A New Approach to Construction and Deployment of Software

Author: Thomas Sewell

Bachelor of Engineering (Software Engineering)

Undergraduate Thesis

Submitted November 2, 2004

Supervisor: Kevin Elphinstone

Abstract

Conventional approaches to software management separate the task of software construction from that of software deployment. However, these systems are very similar in operation. Both track the relationships between software and ensure these relationships are maintained, although there are substantial differences in scale and type of relationship.

Common difficulties in implementing either system arise from the inability of each to communicate directly with the other. For instance, the common task of configuring a build for its environment is only difficult because the construction system cannot access existing deployment information and has to duplicate it. This thesis investigates the design of a single system that can perform both software construction and software deployment tasks.

Through the development of a prototype, this thesis is able to demonstrate the capabilities and advantages of this approach. In particular, the prototype is unique among its peers in being capable of correctly using a cross compiler.

Acknowledgements

I would like to thank everyone who has given me sane and reasonable advice throughout this slightly crazy project. In particular, these thanks go to my supervisor, Kevin Elphinstone, who has kept me on track, but also to all of my friends and family, who have helped to keep everything in perspective.

I'd also like to thank everyone who helped to proofread this document. You all know who you are.

Contents

Chapter 1

Introduction

1.1 Overview

This thesis investigates a new approach to software construction and deployment, that of developing a single system capable of addressing both tasks. There is substantial potential in this approach, but it introduces a great deal of complexity.

This introduction demonstrates how this complexity can be addressed by developing two languages, one of which specifies which software objects play the same role, and another which provides a means of tracking the differences between those objects. The potential advantages of this approach are also demonstrated.

Chapter 2 investigates the background to this project, considering a number of other systems that have addressed one or both responsibilities.

Chapter 3 analyses the problem of designing a language to specify and track the differences between semantically similar software. It motivates and introduces the attribute language that was designed, and demonstrates how a number

of example issues can be tracked using this language.

Chapter 4 analyses the problem of designing a language to specify which software objects play a given role. It looks at some interesting design possibilities, analyses issues related to the semantics of roles, and introduces the provision language that was used in this project.

Chapter 5 moves from language design to the challenges of implementation that were raised during development of a prototype. The problem of developing a builder algorithm is analysed in particular detail. Many serious implementation difficulties are discussed, and some of them are solved.

Chapter 6 describes the achievements of the prototype development, and in particular evaluates the degree to which the prototype demonstrates that certain tasks are feasible.

Chapter 7 discusses the achievements that were made, the extent to which conceptual work for further achievements is in place, and some of the difficulties that were encountered during the project.

Chapter 8 summarises and concludes the thesis.

Chapter 9 discusses the substantial opportunities for further research into this approach.

1.2 On the Nature of the Proposed System

The project aims to develop a single system that simultaneously addresses the tasks of construction and deployment. The first question which must be considered is what such a unified system looks like.

Both tasks principally consist of tracking the dependencies of software. In addition, both tasks are performed in a similar manner, guided by the fundamental rule that all dependencies of an object must be built or installed before that object can be built or installed. Thus a system that addresses both tasks

will be principally concerned with tracking the dependencies of software on all scales, and ensuring that those dependencies are met.

However, there are unique features in each task. Construction management systems, usually called build systems, handle varying transformations. By comparison, deployment systems, usually called configuration system, almost always export these transformations to a build system. Build systems are also usually interested in rebuilding software that has already been built after a small number of sources have changed. Deployment management systems need to deal with more complex dependencies and interactions (such as multiple packages that provide the same services, and packages that conflict with each other). Configuration systems are also frequently interested in the task of retrieving software to be installed from various repositories. Ideally, this project would attempt to implement all of these features. However, due to time constraints, the work will focus on the most essential functionality.

It is absolutely essential that the unified system be able to handle multiple versions of the same software, dependencies on other software, conflicts, and a wide variety of transformations. Retrieving software from repositories and minimally reperforming previous builds after changes are non-core functionalities that could be addressed as separate tasks, and will not be addressed in this project.

This is not straightforward, as these tasks interact. Take for instance the common dependency where an input to the linker uses symbols it does not define (e.g. C files that use the `extern` command). This dependency requires that an additional library or other resource which supplies these symbols be involved at the link step. The standard approach to this in build systems such as Make (see page 16) is to specify the additional link target directly as part of the specification of the link transformation. However, if multiple implementations

for this particular input to the linker are available, and other implementations have different symbol dependencies, it is clear that this approach is not ideal.

There are numerous other issues relating to a particular software object which may affect how it is used, which other objects it must and must not be used with, and whether it can be used in certain environments at all. These will be discussed in detail later.

The particular challenge is that these issues must be tracked carefully if any build system which admits multiple differing implementations is to be designed. In the example above, the input to the linker is probably the outcome of compiling or assembling another source file. Its symbol dependency is thus inherited from that source file. This means that these issues need to be tracked carefully from sources to derivative software.

This idea of issue tracking motivates the fundamental approach adopted in this project. There are two related but separate problems to be addressed. The first is to describe precisely which of these issues affect a given piece of software, and how that information will be passed on to derivative objects. The second is to deal with the problem of multiple versions, and in particular, multiple implementations of the one requirement that are built through entirely different transformations. In this project, these problems are addressed by developing separate but closely related languages for specifying the relevant facts for either problem.

The former language will be referred to as the attribute language, and the latter as the provision language. The nature of these languages and the reason for the names are discussed in the relevant chapters.

It could be argued that these languages are not truly discrete, but rather complementary parts of the one language. The distinction is minor. For the purposes of this thesis, there are two different languages.

Using these languages, the system can handle the tasks required above. The provision language allows us to handle multiple versions of the same software, and also a wide variety of transformations. The attribute language allows dependencies and conflicts to be specified and managed.

The focus of this work will be on developing these languages, and developing a system that interprets and uses them to perform build and configuration tasks. This is the nature of the proposed system.

1.3 Aims and Potential Advantages

The first aim of the work is to develop languages for describing and manipulating the properties and role of a piece of software as described in the previous section. The second aim is to develop a system which can handle the construction and deployment of software objects of arbitrary size by using this language, and prototype it.

However, the objective of this work is not only to demonstrate that this alternative approach is feasible, but also to demonstrate its benefits. There are a number of tasks which are difficult to perform using existing systems but which can, in theory, be easily performed by the proposed system. The third aim of the work will be to ensure that the prototype can in fact simplify these tasks in practice. These tasks are discussed below.

A fourth objective of the project that was not addressed is discussed later, in the section "Extensions to the Application Domain" (see page 79).

1.3.1 Tracking Multiple Versions of Compilers

It has been observed [1] that there is a dependency between a compiled file and its compiler that is difficult to express using conventional approaches. The dependency is significant, as if the compiler version changes it may be necessary

to recompile all of the sources. This problem is only difficult because the build system has no useful conception of compiler version. However, when a single system manages software dependencies at arbitrary scale, there is no problem in assigning a dependency between a file and a compiler. This is done in practice by making the compiled file have as part of its status the version of the compiler that was used to compile it. Any consistency checks that are required can then be made.

1.3.2 Using Legacy Compilers

An emerging problem in configuration management is legacy software which is incompatible with the newest compilers. The heart of the problem is that these incompatibilities are usually well documented at the configuration level but there is no easy way to communicate them to the build system. In practice, the configuration system will recognise that a legacy compiler is needed and have it available. However, if the build system calls on the compiler by its place in the filesystem, with a name like `gcc`, the only way for the configuration system to ensure the correct version is used is to edit the filesystem. This potentially affects other programs, and becomes a difficult task to coordinate. However, with one system supervising both configuration and installation this is trivial, since requirements on compiler version can be present in the build system.

1.3.3 Cross Compiling

Cross compiling is another situation where multiple compilers need to be used together carefully. Cross compiling can be attempted by simply substituting a cross compiler for an ordinary one. However, this technique will frequently fail. Consider the example of a software package that contains a parser, and requires during the building process that this parser be used on some of the source files

to construct certain data structures. In order to correctly cross compile the package, the parser must be built for the host machine. In fact, the parser may need to be built again if it is needed as part of the final package. This is difficult to do using conventional systems (and usually requires that the author of the software has written a separate script or make option that performs the cross compile).

However, when the status of software is being managed, the correct operations can be deduced automatically. The key is to differentiate between the status transformations caused by the cross compiler and the host compiler. What must be understood is that these compilers differ in terms of their target architecture/system/platform, and that these differences are reflected in the compiled files, which can only be used in certain systems. We can then see that a request for cross compilation is in fact just a request about the status of the final object. It can then be automatically deduced that in order to build the data structure, the parser must be compiled and used locally. It is also easily deduced that this version has a status which does not match the request, and thus a new version must be constructed to go into the cross compiled package.

1.3.4 Deducing Large Scale Dependencies

Any configuration system has to track the dependencies of large packages of software. However the dependencies of these large packages can usually be computed with reference to the dependencies of the files in them. For instance, APT (see page 23) evaluates the library dependencies of packages by running ldd on all binaries contained within them.

This approach can be generalised when it is realised that there is a dependency relationship between a header file and the library whose functions it prototypes. A source file which includes the header file will almost certainly

compile to a binary which depends on the library. This is convenient, since to correctly evaluate the dependencies of the source file (the small scale dependencies), it was necessary to list the header file. With appropriate tracking, the dependency on the library generated by using the header can be tracked to the finished executable. Thus the larger scale dependencies of an executable can be deduced from the smaller scale dependencies of its source files. In a similar manner, the large scale dependencies of an entire software package can be deduced by accumulating together all the small scale dependencies of the files in it.

Using the small scale dependencies to identify the large ones is very significant. It implies that the integration between the large and small scale systems is useful in terms of directly sharing data. This goes some way towards justifying this project's approach.

Chapter 2

Background Review

There are a great many configuration systems and build systems which implement a variety of features. However, the purpose of this thesis is not to develop a fully featured build or configuration system. Rather, this thesis is particularly interested in systems that have, to one degree or another, attempted both tasks. Thus a number of build and configuration systems with interesting features are omitted here.

Some of these systems have at least partially addressed both build and configuration management issues. In this case, the most interesting question is how these systems have approached the problem of differences between multiple implementations of the one role.

2.1 Make

Make[2] is the build system that is most widely known. Its simple approach to constructing software is motivated by the common task of rebuilding binaries as source files change. Make is used as a build system in a number of projects, especially free projects. Although Make does provide the convenience of automatic

building, it has a number of flaws.

Make has a programmable syntax which allows, for instance, specification of the "src/*.c" set of files. However, Make does this through setting variables that are closely related to environment variables, which leads to difficulties in using a Makefile out of its original context. The particular problem is that Make allows environment variables to override its internal variables. This allows some customisation, and passing of variables from parent invocations of Make to other invocations they create. However, these variables propagate in an arbitrary and invisible manner, and thus it is very difficult to use Makefiles without being familiar with the manner in which they were written.

In addition, Make's timestamp approach to file construction can lead to problems if the timestamps of the files are manipulated in an unexpected manner. Many projects controlled by Make need an infrequent use of touch when builds refuse to update, especially if the Makefile is being updated. Another problem is that too much may be recompiled. A particular offender in this respect is Sun's Java compiler, which may update the timestamps of class files other than the one being compiled, resulting in extra work down the track.

Finally, in large projects, Make's recursive checking of timestamps can become inefficient. If a programmer makes changes to a single file and calls Make on a large tree, it may take a long time for all the timestamps to be checked.

Despite all of these flaws, Make continues to be used, especially in small projects where Make's lack of scalability is not a burden, and its unexpected behaviour is unlikely to cause major problems. In conclusion, Make usually does what it ought, but there is no guarantee.

Make, and other similar build systems, does not attempt any form of configuration management. It allows no possibility of multiple implementations of the one role, except where one is outdated and to be replaced.

2.2 Children of Make: GNUMake, Jam, Ant, and others

A vast number of projects have been motivated by the problems of using Makefiles for medium to large sized projects. There are three main reasons for this:

- Make's pattern based syntax is frustrating, especially when there are multiple source directories, or the relationship between files and their dependencies cannot be easily expressed with patterns. Better programmable support is universally desirable.
- Make files are very system dependent and low level, using environment variables and shell commands intensely.
- There are many high level programming languages, all of which are far superior to Make for complex tasks. For each of these languages, there is a large body of developers who are enthusiastic to use its features in new environments.

As additional evidence, these programs usually cite "Recursive Make Considered Harmful"[3].

These Make clones include GNUMake[4] (Make with some useful extensions), Jam[5] (Make with the fundamental rules made explicit in an interpreted language), the Make module[6] (Make in Python), Rake[7] and Rubuild[8] (Make in Ruby), Bras[9] (Make in Tcl), Ant[10] (Make with XML and Java scripts for HTML fans), and many others.

The problem of ease of use seems to be reasonably well addressed. In addition, some of these solve the problem of system dependence. By writing commands in interpreted languages rather than with shell instructions, it is easy to avoid the dependence on a UNIX style command set. Some of these

implementations get better performance by running processes in parallel.

However, the focus in most of these applications is simply on removing the problems with Make. They present few major changes to its model of operation, and do not address configuration issues at all.

2.3 Cons and SCons

The Make replacements mentioned above attempt to solve the immediate problems of Make but tend to provide few original features. Cons[11] and SCons[12] are the exception, in that they do provide some extensions, and in particular integrate some external tools that were extensions to Make.

The initial ambitions of these projects are exactly the same as those above. Cons intended to provide a less unwieldy version of Make by using Perl, the most popular scripting language of the time. Python having replaced Perl as most popular scripting language, the SCons project initially set out to reimplement in Python. However, a number of significant improvements were made in these projects.

In particular, the Cons system is designed to allow builds spread across multiple directories and build specification files. It uses MD5 cryptographic hashes to detect whether files have changed, a significant improvement over Make's timestamp approach. In addition, searches files for indications of their dependencies. For instance, when compiling C files, Cons searches for `#include` lines which indicate further dependencies. SCons provides similar features, but is more extensible.

In summary, these are excellent tools. Not only is Make's syntax replaced with popular alternatives, but the overall Make approach which was full of problems is converted into a robust and reliable system. However, most of these improvements are tangential to the ones proposed in this thesis.

What is of interest is that these systems present a mechanism by which the arguments and inputs to a compiler or other program can be specified independently of the format in which the relevant program requires its arguments. This allows builds which are portable across multiple compilers. It also allows compilers to be swapped as necessary.

A similar system is needed in this thesis. However, Cons and SCons provide no means of interpreting the differences between said compilers.

2.4 Maak

The author of "Integrating Software Construction and Software Deployment"[1] proposes a system called Maak. The purpose of Maak is in theory to make deployment transparently part of the build process. It does this by providing a Make style system for building a package from its sources. Deployment is provided by allowing sharing of derivative files. This feature is not fully implemented yet, but the general approach is that if a build is being considered and an identical one has already been completed (by another user of the system, or possibly by a peer site) it can be used instead.

Thus deployment of binaries becomes merged with deployment and building of source, which has some aesthetic appeal. However, this approach is not demonstrated to have any especial merit.

The heart of the proposal is to make deployment merely a shortcut in construction. There are a number of shortcomings with this approach. Commands are issued to Maak in terms which closely mirror its internal language, requiring the administrator to understand Maakfile internals. All file names are replaced with hashes based on what they were built from, thus cannot really be used without consulting Maak. Finally, the author of the paper acknowledges that one of the problems he wanted to solve, the listing of dependencies of files on

their compilers, is impractical to solve in Maak, as it will recursively check whether the compiler needs rebuilding every time the file is recompiled.

However the critical flaw lies in the assumption that a binary program built on one machine can simply be transferred to the next. As argued in this thesis, although this binary is semantically the same as the one that would be built on the local machine, there are a number of potential differences which need to be carefully tracked.

In other words, Maak introduces a situation where multiple possible results of the one build are inevitable, but introduces no mechanism for differentiating between those possibilities.

2.5 Vesta

Vesta[13, 14] is a configuration management system produced at the Compaq/Digital Systems Research Center. It was designed to be used by a large software development group. It is thus designed to be used intensely for large builds. Vesta is an all in one solution for software development, and although it is called a configuration system the emphasis is on the built in version control and build systems. Vesta uses the integration between these systems to achieve a number of additional efficiencies and guarantees. One such efficiency is that Vesta's build system does not check timestamps in the Make manner, instead depending on the version system to know what has updated, and proceeding from there to the derivatives.

Vesta's main advantage is guaranteed repeatable builds. Vesta keeps a complete log of all the versions that are used in building software, including compiler and library versions, allowing any build to be recreated and any test to be rerun. This is extremely useful to a software development team trying to track where bugs arose from and whether they have been addressed. Vesta also uses low

level features of Linux to trace which files a compiler or other program opens, and can thus deduce exactly what the dependencies are.

Vesta is a powerful system for large scale software development projects. However, it is much too complex to be used as a configuration system in most other environments. The complexities involved in using Vesta require a level of technical expertise that puts it out of range of the general public. From an academic standpoint, however, the Vesta approach is extremely interesting because of the advances it contains.

Vesta has an interesting approach to the problem of multiple versions. It effectively prevents them. By ensuring that the entire source and environment involved in construction of a resource is known, Vesta makes it impossible that there are any differences to track. If such homogeneity was possible in less controlled environments, this thesis would be unnecessary. However, most deployment problems require installation onto heterogeneous machines, a problem which Vesta does not address.

2.6 Red Hat Linux: RPM

Red Hat's Package Manager[15] (RPM) is the most commonly seen package handler for Linux, and its packages (usually `.rpm` files) are provided by almost all free projects. In addition to containing all the files that are required for installation, RPM packages incorporate the scripts required to build and install the software, and a list of which files in the system will be installed as a result of the installation process.

The RPM utility coordinates installation or removal of packages that the user has manually acquired. RPM checks that all dependencies of the package being installed are installed. If they are not, however, the user is required to manually amend this.

RPM has received a number of upgrades recently to make it comparable in usefulness with the Debian equivalents. For instance, `rpm -i *.rpm` (install all RPM packages in this directory) now understands to shuffle the order of installations in order to get dependencies right.

The most significant remaining flaw in RPM is the way it handles conflicts. RPM does this through knowing which files are associated with which packages. Any packages which would supply the same file are considered in conflict. However, RPM provides no way of communicating that multiple packages provide essentially the same service. The example the author is familiar with relates to the SDL and libSDL packages. SDL provides a full development package for the SDL library, whereas libSDL provides only the shared library files. These packages will always conflict. In addition, if other installed packages call for libSDL, RPM will attempt to block its removal, even to pave the way for SDL.

RPM is the first significant configuration system. However, as noted above, it does not deal especially well with any multiple versioning.

2.7 Debian Linux: APT and `.deb` packages

Debian Linux[16] provides a package management system which is a substantial improvement over RPM. In addition to listing dependencies which must be installed first, Debian packages can list negative dependencies. These packages are ones which conflict with each other, and should never be installed together. Explicitly announcing conflict, as opposed to the RPM model of deducing conflicts from the list of files provided, allows detection of more subtle conflicts. For instance, having two installed HTTP servers at the one time would create a conflict not for a file, but for port 80. In addition, Debian packages may list that they implement another package, thus allowing multiple implementations of the same resource to be available to Debian users, and to be interchanged

without the problems this would create using RPM.

On top of this system Debian provides an extremely useful suite of software under the name APT (Advanced Package Transfer). APT allows users to request the installation of any package, including ones that have not been retrieved yet. APT will automatically retrieve the package, and any dependencies, from known archives, and will then pass the retrieved packages to the lower level installation system in order. This feature means installation of software can be done without any user supervision or understanding of the process, one of the most significant advantages of the Debian Linux distribution.

APT, like Cons and SCons, is an excellent system in its field. However, its features are tangential to the particular inquiry of this thesis. Although it can consider a number of kinds of conflicts, provisions and other relationships, it has no need to study these relationships through a build graph.

2.8 APT beyond Debian

There can be no doubt that automatic installation of software is universally desirable. Thus it comes as no surprise that a variety of implementation teams are working to extend the advantages of APT to other users. A number of projects have ported APT in one form or another to work on RPM archives[17]. This means that users of the various distributions that are designed for RPM can use APT's automatic services.

More ambitious implementors are going further and trying to bring together the package databases, allowing users to simultaneously use RPM and APT packages alongside each other. The GNUpdate project[18], which promises all this and a handy GUI, is apparently making steady progress.

In addition, the Fink team[19] are bringing Debian tools to Mac OS X users. This project is a significant one for Mac OS X, but will provide no functionality

other than what is available to Debian users.

2.9 FreeBSD: Ports

FreeBSD[20] develops a strikingly different software deployment system. Unlike most package management systems, the ports tree represents a single large file tree, with Makefiles in the tree capable of coordinating retrieval, construction and installation of packages.

The name of this tree is not a coincidence, the contents of the tree being literally the collection of available software that has been ported to BSD.

Evangelists of the ports approach point to the relative advantages of source distribution over binary, including independence of exact names of filesystem libraries, the capacity to supply compile-time tweaks, the capacity to make small changes to harmonise the package with the system, etcetera. However, FreeBSD also provides, and seems to recommend highly, a system for maintaining a collection of packages. These packages are more or less exactly precompiled parts of the ports tree.

An interesting artifact of this approach is that the configuration and build systems were closely integrated in the original ports tree. The Makefiles which coordinated the installation of software effectively performed both build and configuration level tasks. However, from the fact that the package system was developed, and that it is recommended over using the ports tree directly, it seems that the Makefiles were inadequate. This is probably not a surprise, given the inability of Make to cope with relationships other than direct dependencies.

Other versions of BSD[21, 22] make substantial modifications to these systems. However, the fact that the ports tree is effectively a filesystem instantly disallows the possibility that multiple versions of the same file might be managed.

2.10 Gentoo

Gentoo Linux[23] is one of the newest major Linux distributions. Combining ideas from BSD style ports trees with Linux approaches, the authors of Gentoo developed a fairly complex configuration and build system. As a result, Gentoo has become popular, especially among users who are interested in tweaking their compiles for performance.

It is interesting to note that the authors of Gentoo have discovered some of the issues addressed in this thesis. In particular, the Gentoo system allows multiple versions of software through its "slots". These slots allow different versions of the same software to be installed. However, this system does not yet support installation of multiple software with the same version. It is mentioned that when such functionality is available it might be used to install a cross compiler.

This is fairly significant. A mainstream Linux distribution is moving in the direction of supporting multiple versions of the same software for such purposes as cross compilation. If this is ever done, a system for tracking accurately the differences between these systems such as developed in this thesis will be necessary.

Chapter 3

Designing a Language to Specify Attributes

This chapter covers the design of the attribute language, a language which will be used to track various issues about pieces of software which restrict the way in which they may be used.

By tracking dependencies and conflicts in software, and in particular by tracking these attributes as the software goes through a series of transformations, this language allows the system to effectively deal with these tasks.

3.1 Attributes

When tracking the status of software, it is desirable to track a large variety of concerns in parallel. One such concern might be the compatibility of a given piece of software with older versions of a particular library. Another might be the level of compiler optimisation it has received. Both of these issues are worth tracking, and they are clearly entirely separate issues, thus they are tracked as

parallel problems. More generally, all issues are tracked as a series of parallel problems. For each of these separate issues that concern a given software object, a small amount of information about the object which is relevant to this issue is kept. This information is called an attribute.

This is the first insight into the nature of an appropriate language for tracking software status. Such a language will be interested in describing a series of attributes for each of the pieces of software in the system. This is why the language designed here is referred to as the attribute language.

3.2 Entities

The concept of a "piece of software" is a fairly cumbersome one. A more appropriate atomic unit is an entity. An entity consists of a software object together with a series of attributes which describe it. The majority of entities will consist of a single file together with a series of externally specified attributes. More exotic entities might consist of a series of entities together with a different set of software, such as a number of files, or one fragment of a file.

The advantage of thinking about our tracked software in terms of entities is that it better reflects the approach that is taken. The system is principally concerned with the attributes of the entity. The internal software will be used, built, or replicated when the attributes are appropriate. Furthermore, the system will never manipulate any software without reference to the entity it is in.

To summarise, the system uses entities in an atomic manner, and the thesis will also.

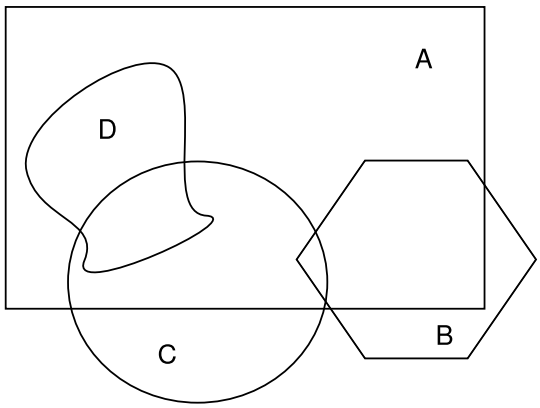


Figure 3.1: Sets $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ and $\mathbb{D}$

3.3 On the General Structure of Attributes

A contrived example helps in motivating the general structure of the attributes. Suppose that there is a compatibility issue where certain binary code can only be deployed on certain machines. The exact issue is left unspecified, although there are many examples of compatibility attributes which will be discussed later. Suppose also that we have four potential inputs to the linker, A, B, C and D, and that because of this issue the code in A can be deployed only on the set of machines $\mathbb{A}$, the code in B only on $\mathbb{B}$, etcetera.

Consider the effect of linking A and B to produce AB. We expect that whatever issue causes A to be deployable only in $\mathbb{A}$ will also effect AB. Likewise, the AB will be restricted to $\mathbb{B}$. When these are taken together, we see that AB is restricted to $\mathbb{A} \cap \mathbb{B}$. Likewise BC[1] is restricted to $\mathbb{B} \cap \mathbb{C}$, ACD to $\mathbb{A} \cap \mathbb{C} \cap \mathbb{D}$, etcetera.

This structure, with a series of intersecting sets, neatly describes the behaviour of compatibility attributes. However, it is clear that it is not the exact contents of these sets that is interesting. Rather, it is the fundamental struc-

[1] This is not impossible. AB and BC are themselves potential inputs to the linker, produced with a partial link.

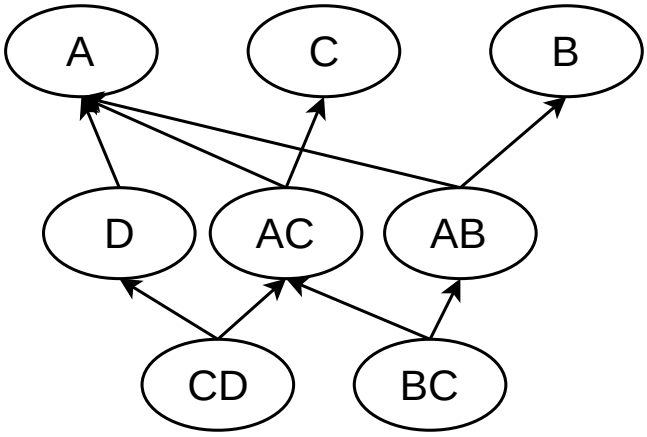


Figure 3.2: Partially Ordering on $\mathbb{A}$, $\mathbb{B}$, $\mathbb{A} \cap \mathbb{B}$, etc

ture obtained when taking their intersections. This structure is well known in formal mathematics as a lattice, a special case of a partially ordered set. The mathematical study of such sets is interesting but irrelevant, and further details are omitted here[2]. Fortunately, all finite lattices can be represented in terms of intersecting subsets. Thus the intuitive knowledge these sets and subsets give us is not eclipsed by the mathematical concept. Rather, the mathematical concept identifies the exact rules that our intuitive knowledge.

Furthermore, these simple rules give us a framework on which to implement a series of heterogeneous attributes should have. All an attribute need define is a partial ordering and a greatest lower bound operation. Here, the ordering is the subset relation, but more generally it is any relation that is transitive, reflexive and antisymmetric. Here, the greatest lower bound is the intersection, but more generally it is any appropriate operation.

This interface is simple and can be widely implemented. What remains is to demonstrate that most of the issues that need to be modeled can be accurately

[2]See http://en.wikipedia.org/wiki/Partially_ordered_set for a real introduction to the field of partially ordered sets.

represented using such lattices.

3.4 Compatibility Attributes

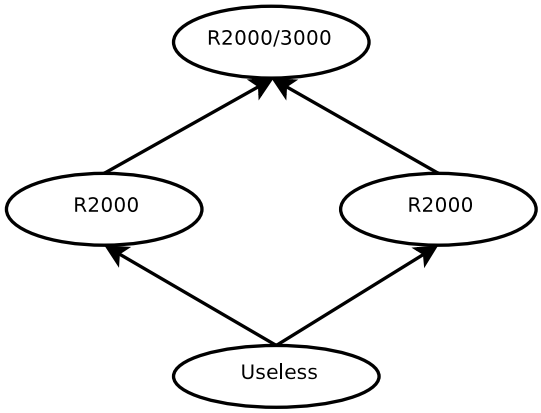
This class of attributes determines exactly which set of environments a certain entity and its derivatives can be used in. There are three different subclasses with different implications.

3.4.1 System Compatibility Attributes

System compatibility attributes determine exactly which systems an entity can be used on, due to compatibility issues with fundamental parts of those systems. The semantics of these attributes are exactly those discussed above.

One example is the architecture compatibility attribute. This is the attribute that determines which architectures a file can be used on. The set of values this attribute can take will include a variety of architecture types (for instance i86, sparc, alpha, ppc, mips, etc). In addition, it is possible to have degrees of specification. For instance, the MIPS R2000 and R3000 processors are sufficiently similar that it is possible to write binaries that work on either of them. It is also possible to write binaries that work on just one. Similar distinctions exist for most families of processors.

Now consider the task of linking two entities, one of which will work on any MIPS, and the other only on the R2000. It is clear that the resulting entity works only on the R2000.



In this case, our relation takes the form mipsr2000 $\supseteq$ mips as code that runs on the R2000 is a superset of code that runs on any MIPS.

Another class of system compatibility attributes relate to system call specification. For instance, most modern operating systems support the `select` system call, which allows a process to sleep until either a timeout elapses or one of a number of events occurs. However, if such an event cuts the sleep short, the Linux implementation of `select` modifies the time structure it is given to reflect the amount of time that was not slept. Most other implementations of `select` leave the structure unmodified. Thus there is a compatibility problem. The vast majority of code does not use `select` or makes no assumptions about the time structure after the call. Some code, however, reuses the time structure, assuming either that this results in the same length timeout or that this results in a timeout at the originally scheduled time.

3.4.2 Local Compatibility Attributes

In the same vein as the issue of system incompatibility, a common incompatibility issue is when two entities which are potentially used together are incompatible.

Most examples of such incompatibilities are of little note, but most large

projects involve them, particularly where interfaces have changed through time. An artificial example will have to suffice. Suppose that a project uses a utility library U. Suppose that versions U_1 and U_2 of U are similar, but version U_3 implements an updated interface which is incompatible with the old one.

Now suppose that there are a number of different applications which use this library. These applications need to be partitioned into those that assume the new version and those that assume the old. These two states are represented by a local compatibility attribute. The effect of combining software should also be fairly clear. If two entities with differing assumptions are combined, the outcome is an error value.

3.4.3 Dependency attributes

There is a class of compatibility requirements which specify properties of the environment which the system has the power to change. Suppose we have a program W which has been written in C, and compiled with a standard compiler. There is a qualification on the environment in which W can operate. W, like most programs compiled with standard C compilers, depends on the `libc.so` shared library. In its absence, W is useless. However, unlike previous compatibility problems, the system is in a position to affect whether this problem exists.

This situation is typical of a class of attributes that form dependencies. Each of these attributes specifies that the entity depends on additional resources. There are two variants. Permanent dependency attributes specify that an entity requires the additional resources permanently installed in the system in appropriate locations for the entity to be useful. Immediate dependency attributes are common where entities are used as inputs in the creation of other entities, and all that is required is that the additional resources be supplied

when the entity is used as an input.

There are many examples of dependency attributes, for instance:

- A source file in a language like C or latex which depends on a header being present at the time of compilation.
- An object file that requires symbols to be supplied from a common library at the time of linking.
- A program which executes a shell command and thus assumes the permanent presence of another utility program in the executable search path.
- A script which begins ‘‘`#!  /usr/bin/perl`’’
- A program which assumes the permanent presence of certain data files in a predetermined location.

These dependency attributes will be the principal method of addressing configuration management concerns. The intrinsic assumptions about the layout of the system which bring about the configuration problem can be encoded into these attributes. What remains is to ensure that entities are not used except when dependency attributes are met. The main configuration management task will be to ensure that all dependency assumptions are met. Managing this task from the point of view of meeting assumptions yields the advantage that no canonical location for any file must be adhered to. For instance, if a program is initially needed in `/usr/bin`, and a second program is added to the system which depends on the former being in `/usr/local/bin`, this new requirement can be met on the fly. This is a significant improvement over approaches which require an installed program to have one or more fixed install locations which must be used by all other software.

3.5 Descriptive Attributes

In contrast to compatibility attributes, descriptive attributes are used to encode information about an entity that does not affect which environments it is compatible with. Such attributes have no special significance to the system, but may be of interest to the end users.

Presented here are some common examples.

3.5.1 The Compiler Optimisation Attribute

This attribute encodes the level of compiler optimisation that was used in building certain entities. It is especially useful, because it means users can express their preference for entities which have been compiled with a certain level of optimisation. Furthermore, this request is expressed not in terms of overriding some configuration information (like a `CFLAGS` variable) but by making a request about the attributes of the final object.

The level of compiler optimisation is conceptually some kind of linear progression, starting with no optimisation and progressing to a hypothetical 100% optimisation. Since this is an ordered set, our partial order is obvious. Combining two entities takes the minimum level. The difficult task is mapping the compiler optimisation levels offered by various compilers onto a uniform scale, a challenge which is left to future work.

3.5.2 The Language Output Attribute

This attribute simply defines in which languages the program may produce output. This may sound fairly obvious, but is a meaningful question, especially where error output is concerned. Suppose a program is written in French, and compiled against a number of standard libraries which report errors in English. This means that some situations may cause users to be faced by errors they do

not understand. This is a potential nuisance.

This attribute allows this issue to be tracked. In this case the attribute in the original program entity would record that the output was in French only, and likewise the attribute for the libraries would record that they outputted English. It should be clear that the effect of combining two entities takes not the intersection but the union of the sets. Thus it was known that the final entity would output in both French and English.

The nice thing about attributes of this nature is that they allow special interest groups, such as speakers of other languages, to use exactly the same systems. Such groups simply impose restrictions on the attributes of the entities they will accept.

3.5.3 License Tracking

One issue which has absolutely no bearing on the function of any software but which is fairly significant to end users is the licensing of various software.

Common licenses in the free software community include the GPL, LGPL and various BSD-style licenses. Proprietary software is also supplied with a variety of end user license agreements. These agreements in nearly every case impose some restriction on how the software can be legally used. In some cases, these licenses also make significant legal demands of the user, such as the forgoing of substantial legal rights.

When software is supplied that may be combined with other software in one manner or another, these licenses often stipulate that they continue to apply, in one form or another, to the aggregated object. For instance, the GPL applies directly to such works, whereas the LGPL applies a different set of requirements.

Such legal restrictions could in theory be tracked with one or more attributes. For instance, the application of the GPL to all derivative works can be simply

modelled with a binary attribute, true or false, which unifies according to the logical "or" operator.

This example is interesting, and is included here to illustrate the scope of the language to address real world concerns. However, the language does not quite fit this problem. The basic issue is that the relevant license contracts are much more complex than a simple computational model. Often, a number of conditions on the aggregation are made which the attribute system cannot replicate. For instance, the GPL explicitly specifies that aggregation of one work under the GPL with another in a storage or distribution medium does not extend the GPL to the entire medium. The attribute system does not provide any means for comprehending these exceptions.

Thus, with the attribute system as designed at the moment, tracking this particular issue is not particularly practical. However, the language is almost sufficient, and if this functionality was useful, appropriate extensions could be made.

3.6 Effect of transformations

The attributes of a derived entity created in some kind of transformation are not derived entirely from the inputs. Information about the transformation itself is also important. Consider, for instance, the action of a compiler. The compiler will target a particular architecture, and thus the outputted code will have that restriction, as well as any other restrictions intrinsic to the input. This would appear to match our case above, where we take the greatest lower bound of the possibilities for the compiler and those for the starting source. For reference, it is possible, but unusual, for source code to be architecture specific. Considering that the same greatest lower bound calculation as elsewhere is used, it ought to be asked whether there is any difference between linking and compiling at all.

However, recalling our example of a cross compiler we see that it is undesirable to use the compiler's architecture information directly. Rather, we would like the compiler to have a separate attribute that controls its target architecture, which is mapped into the architecture attribute for compiling.

More generally, we want our compilers and other active components to have two sets of attributes, one which describes the program itself, and another which describes the imparted attributes left by the program's action.

It would appear the issue of computing the effect of transformations on attributes is completely resolved. However in more general examples, more complex systems are needed. Consider the example of an optimising compiler. Intuitively, this compiler grants to the code it compiles something it did not otherwise have. The system of taking greatest lower bounds cannot implement this, as the greatest possible outcome for a file being transformed is the same state as it began in.

This is a substantial problem. There are a variety of possibilities. One which solves this case is to imagine all sources as having the best possible optimisation, and losing it when run with non optimising compilers. This works for most cases, but depends on the fact that the compiler must be run (so that optimising compilers have optimisation essentially because all other compilers have reverse optimisation). In the case where an optimiser is a separate program to the compiler which is run optionally, this approach fails. Another option is to allow the computation of lowest upper bounds instead of greatest lower bounds for certain attributes, which works in this case, but may fail in others.

Essentially, to handle the most general case, the effect of the transformation on certain attributes must be explicitly programmed as part of the specification of the transformation. This approach was accepted for use in this project, but it is likely that a better method can be found.

3.7 Final Design

These considerations have established the fundamental structure of a language. In summary, these points specify the syntax of the language:

- The language is constructed from a variety of attributes which have their own individual properties.
- Each attribute has a name (a unique identifier).
- Each attribute has a simple set of values.
- Each attribute supplies code that defines the partial ordering on its values, and a means of finding a greatest lower bound.
- Each attribute also supplies code that defines a method of interpreting its values. This may require a parser, or if an appropriate preparser is available, some kind of constructor.
- Each attribute also supplies a default value.
- The syntax of the language is used when specifying an entity.
- Each entity has an appropriate value for every attribute.
- An entity specification contains firstly a filename or other reference to the actual software contained in it, and also a series of attribute specifications.
- Each attribute specification contains the name of an attribute, and data which is interpreted as mentioned above to set the value of an attribute.
- Each attribute not specified in an entity specification is assumed to be of the default value.

In addition, these points specify the semantics of the designed language:

- The output entity of a transformation has its attributes defined by the attributes of the input entities, and various properties of the transforming program.
- For some attributes, the relevant value is determined directly by the behaviour of the program.
- For all other attributes, the relevant value is the greatest lower bounds of the relevant values from all the input entities and the imparted attributes of the program.

Chapter 4

Designing a Language to Specify Roles

This chapter covers the design of the provision language, which is used to specify how an entity is useful to the system, that is, its role. By determining which entities have the same role, this language handles the issue of multiple versions of the same software.

To completely understand the role of a piece of software is well out of the scope of this project. What is required is some way of establishing the role of entities sufficiently precisely that they can be called upon by other entities.

As will be demonstrated in the body of this section, the roles can be represented simply by names in some name space. The function of the provision language is then to correlate entities (including entities that have not yet been built) with these roles.

4.1 Motivating Example

The constructs examined in this section are complex and only vaguely similar to better known systems. A motivating example aids in understanding how everything fits together.

One of the issues this project aimed to handle (see "Aims and Potential Advantages") was managing multiple compilers for the same language. This example is the smallest possible case of this problem. There is a simple programming language L in which certain programs can be expressed in a portable way. These programs are typically compiled into executables to be used. There are two implementations of the compiler available, and both are written in L. They are shipped as the single source files X and Y. These compilers are not completely interchangeable. Although used in the same way, X produces executables which run on platform $\mathbb{X}$, and likewise Y produces executables for $\mathbb{Y}$. To avoid a bootstrapping problem, we also have Z, a copy of X that has been precompiled (somehow), and runs on $\mathbb{X}$. Finally, we have some other application written in L, which is supplied as the source file W.

4.2 Roles

We hope that our language will be able to intelligently pick between L-compilers. However, before we look at how this decision can be made, we first need to establish that there is a decision to be made. An L-compiler is needed in compiling W. Clearly we need some way to specify that an L-compiler is needed. Furthermore, we do not want to prescribe which L-compiler (we hope to decide). Nor do we wish to describe the entire set of L-compilers whenever one is used. In other words, we would like some name or handle to refer easily to the set of L-compilers.

Roles are these handles. A role is precisely a name by which we refer to a set of software that somehow shares a purpose. In any situation where we would like to be able to refer to some entity without specifying exactly which one, we create a role to use. We then specify elsewhere that certain desired entities play this role.

The simplest possible example is multiple versions of the same software. We hope to be able to use things without describing whether we want version 1.1 or 1.2. Furthermore, if a new version 1.2.1 is released, we hope to use it even in software that was released before 1.2.1 was known. Thus we create a role that describes all of the versions of the software, including future ones.

Another common example is where there are many different implementations of a library which all share the same specification (i.e., provide the same functionality). In this case, the various implementations ought to share a role that represents this functionality.

Significantly for this project, any two different objects built from the same source (through different means) will share a role. Suppose we use Z to compile Y, producing Y_Z, then use Y_Z to compile Y again, producing Y_{Y_Z}. Now Z, Y_Z and Y_{Y_Z} are all L-compilers, that is, they all share a role that represents L-compilers. Not that Z and Y_Z run on $\mathbb{X}$, whereas Y_{Y_Z} runs on $\mathbb{Y}$, thus they are far from interchangeable. However, they all have the same semantic function.

Finally, not all roles are shared. For every entity, there is a role that uniquely describes that entity. We have already established that we use roles whenever we wish to refer to entities without specifying which of some set. With this in mind, we can extend this. We use roles whenever we wish to specify one or more entities.

4.3 Packages

Software is installed on the system through packages. A package is a group of files plus some metadata that describes them.

The metadata needs to specify how the software in the package fits into the wider scheme of things, and what restrictions there are on its use. For instance, a package containing Z needs to specify that Z is an L-compiler, and that Z both runs on and targets $\mathbb{X}$. More generally, the metadata needs to describe the roles of the entities in the package, as well as their attributes.

The metadata, then consists of two things. Firstly, a number of "provisions", detailing how the package provides implementations of a certain role. Secondly, descriptions of all the entities in the package, including complete descriptions of their attributes.

4.4 Providing Roles

The principle divergence between a type system and our role system is that type systems are passive. They determine the types of entities which are explicitly generated by programmers. The role system, however, will be used actively. The user may request an implementation of a given role without supplying any information about how to go about such a request. To meet this need the system must keep track of a set of implementations it can consider when attempting to meet such a request. We will refer to this as the "candidate set" of the role.

It is the "provision" statement in the package metadata that allows us to define the contents of the candidate set. Each provision statement introduces one or more elements. The candidate set for any given role consists exactly of all the elements introduced in this way in all packages installed on the system.

4.4.1 Direct Provision

The package which contains Z must explicitly state that Z is an L-compiler. It does this through a direct provision.

These provisions specify that a role is directly implemented by an entity in a package. The effect on the candidate set is simply to introduce this entity.

As a side note, it ought to be mentioned that not only can one role be implemented by many entities, one entity might provide a number of roles, if there is more than one way of using it.

4.4.2 Indirect Provision

Not all of the entities available to the system are provided directly by packages that contain them. There may be entities which are not immediately available, but which can be obtained on demand.

For instance, our system has more than one L-compiler available. Although Z is the only L-compiler that is usable in its current form, X_Z and Y_Z are alternatives, and it is crucial that the system understands this.

For the system to appreciate that X_Z plays the same role as Z, some sort of relationships between the roles of X, Z and X_Z must be drawn. These relationships form something like a type system. Three approaches to this are discussed here, starting with an intuitive model drawn directly from type theory and moving towards a less elegant but more general approach.

Understanding the effect of transformations on objects is not a new problem. The type theory of functional languages addresses this issue. Suppose Z is understood to be of type $T_1 \mapsto T_2$, where T_1 is the type of X and Y, and T_2 is the type of L-compilers. Then clearly $Z(X)$ has type T_2, that is, X_Z has the role of an L-compiler. This initial approach allows Z to be given a meaningful type. However, the types of Z and X_Z are not the same, unless $T_2 = T_1 \mapsto T_2$,

which is something of a contradiction. The other complication is that the given type for Z implies that it only operates on the sources of L-compilers.

It is clear that whatever type is given to L-compilers must transform the L-source of anything into that thing. Type theory addresses this problem with parametric types. In this second approach, an L-compiler has type $Lsource(T) \mapsto T$. It is now clear how to type our entire system. Z, X_Z, Y_Z etc have type $Lsource(T) \mapsto T$. X and Y have type $Lsource(Lsource(T) \mapsto T)$.

This seems to be fairly complete, and even elegant. However this approach does not generalise particularly well. The only reason it worked so well here is that the L-compiler always converts exactly one source to an output. Consider the case of a linker step, where inputs A, B and C need to be linked to produce an output of type T. To start with, the linker can be used on any number of inputs, which means that a clear type for it cannot exist. But more importantly, we cannot assign A, B and C the same type, or the system will assume that linking A, A and A is meaningful. We could give them types like $FirstObjectOfThree(T)$. However, if a different implementation of T had three objects, the system would assume that it could exchange them.

The problem here is fundamental. There is no possible way that the role of A can somehow be extrapolated from T. This is because information about the functionality and responsibility of A disappears at the link step. A might contain unique functionality available nowhere else, or be a standard library used elsewhere. The possibilities are endless, the point is that the role of A needs to be externally specified, and cannot be required to somehow match with the roles of the linker, B and C in a neat functional way.

It is time to introduce a final method which works. Consider above that the linker is not the function, the active component in the system, but rather one of the inputs. There exists a function that takes A, B, C, and a linker, and

produces an entity of type T. In this different approach, the input types are arbitrary. Thus the roles that A, B and C must fulfill to be used in this way can be set appropriately. Thus the system will consider alternative combinations if and only if they are appropriate.

This is the approach selected, and thus the details ought to be mentioned. The indirect provision statement must supply a function, from some number of inputs (one of which will always be an active entity) to an output. Each of these is specified by role. This function is fairly lightweight, essentially just naming which inputs go where, and what additional arguments are needed, in passing to the program. The effect on the candidate set is to introduce a new element for every combination in the Cartesian product of the candidate sets for the input roles.

This requires an example. Suppose in our system we have the L-compilers Z and Y_Z only. Also suppose that we have W, and also W', a copy of W with some minor changes. Z and Y_Z share the role $LCompiler$, while W and W' share the role $AppSource$. Finally, we have an indirect provision, whose function is $f : (LCompiler, AppSource) \mapsto App$. This means that the candidate set for App is now $\{f(Z, W), f(Z, W'), f(Y_Z, W), f(Y_Z, W')\}$.

4.4.3 Provision through Extension

Another kind of provision statement works by extending an original role to a new role. This is the equivalent of declaring a subtype. The new role, or type, somehow extends upon the original, and thus has its own candidate set. However, all elements of the subtype are also elements of the original type. As in object oriented languages, this allows the new type to extend an earlier specification with new features while still supporting the original.

For example, consider the language L_2, an extension of L. Suppose X_2

is a new version of X that implements the L_2 extensions. X_2, and all L_2-compilers, double up as L-compilers. We could specify explicitly that X_2 is both an L_2-compiler and an L-compiler. However, this is tedious, and if development continues to L_8, will probably be omitted. Specifying that the L_2-compiler extends the L-compiler need be done only once, and then can be avoided.

The effect on the candidate set of the original role is to introduce the entire candidate set of the extended role.

Furthermore, one role can extend multiple other roles. Suppose a problem in the design of L was located, and a patched version L' released. Then L' and L_2 have diverged, with L_2 providing new features but L' being more reliable. Neither is a substitute for the other. However, an L_2' might extend both of these and end the divergence.

Multiple extension also allows a role to simply extend any aliases it might have, and thus manage them appropriately. This technique does, however, require that some decision be made as to which role is directly instantiated, and which are aliases.

4.5 Specification of Designed Language

Many different aspects of this language have been considered in this chapter. In summary, these points specify the syntax of the designed language:

- The language is used to specify packages.
- Each package specification consists of a series of provision statements.
- Each provision statement is either a direct provision, an indirect provision, or a provision through extension.
- Each direct provision statement includes the name of the role provided, and an entity specification (see previous language specification, page 39)

- Each indirect provision statement includes the name of the provided role and the names of the set of input roles. It also includes a function which defines how these inputs are mapped to the output. The exact means of specifying this function is left unspecified here, because the approach used in the prototype is somewhat unwieldy, and a better approach has not been developed at this stage.
- Each provision through extension includes the name of the provided role and the name of the extending role.

These points summarize the semantics of the language:

- The candidate set for each role is exactly the union of the sets introduced by each provision for that role.
- Each direct provision introduces a singleton set containing the entity declared.
- Each indirect provision introduces the image under the supplied function of the Cartesian product of the candidate sets of the input roles supplied.
- Each provision through extension introduces the candidate set of the extending role supplied.

Chapter 5

Challenges of Prototype Implementation

Much of the work in the second session of the project went into developing a prototype which implements most of the features of the two languages. This section discusses the challenges of this implementation, and the additional design work that was done in order to make the implementation successful.

5.1 Building

Using the languages developed previously, the system can track a variety of software entities provided both directly and indirectly. However, an indirect provision of some role is only useful until such time as that role is needed. The builder has the responsibility of establishing direct provisions from indirect ones as necessary.

The builder algorithm was the most significant new conceptual work in the prototype implementation. Although a lot of implementation work went into

implementing the two languages and specifying some examples with them, this work was mainly for the purpose of refining and testing the languages rather than developing new ideas. Thus this section will focus on the difficulties of building and the builder algorithm that resulted from them.

5.2 Simple Approach to Building

At first glance, the build problem seems relatively straightforward. An indirect provision supplies a function $f : (R_1, R_2, \ldots R_n) \mapsto R$. The obvious approach in obtaining a concrete provision is to recurse to locate implementations I_i of the R_i (building as necessary), then produce $I = f(I_1, I_2, \ldots I_n)$, an implementation of R.

This approach is fairly simplistic, but is employed to good effect in a number of system. For instance, Make and its immediate derivatives employ this approach.

5.3 Attribute Requirements

Needless to say, the above approach is insufficient. Recall that Z, Y_Z and Y_{Y_Z} are all L-compilers, however they have substantially different attributes. Z and Y_Z run on $\mathbb{X}$, whereas Y_{Y_Z} runs on $\mathbb{Y}$. In any real life situation the user will require not just any L-compiler, but one which runs on the user's machine. Thus the system will always restrict its search to one of these searches.

However, just as any entity might be incompatible with certain environments, any two candidate inputs might also be incompatible. That is, having generated implementations I_1 and I_2 of R_1 and R_2, it may be discovered that $f(I_1, I_2, \ldots I_n)$ is useless because I_1 and I_2 are incompatible. This might be because they make conflicting assumptions about the interface between them,

external resources, or any number of other issues.

This gives us an issue of semantic confusion. The candidate set of R contains $f(I_1, I_2, \ldots I_n)$, despite the fact that this entity is never useful, and probably cannot be constructed. Furthermore the candidate set for L-compilers contains elements that are useless in environment $\mathbb{X}$ or $\mathbb{Y}$. There is nothing wrong with this, aside from the fact that the candidate set is loosely defined. There is no guarantee that an entity in the candidate set of R is a useful implementation of R. The assumption that can be made is that an entity in the candidate set of R is a usable implementation of R in some environment if it possesses appropriate attributes.

Recall that combinations of software cause a lowest common denominator style combination of attributes. Unifying incompatible entities will produce a "bottom" attribute. To exclude this possibility, and to ensure that the built entity meets other requirements, such as compatibility with a given environment, the builder is always supplied with both a role and an attribute requirement for any given entity.

From this addition comes a more appropriate builder algorithm. The builder maps the requirement on the attributes of the output to other requirements on the attributes of the inputs. It then recursively seeks potential inputs, since it has a new requirement and a new role for them. This produces a sequence of candidates for each input. From this is created a sequence of candidates for the original problem. The builder then iterates through this sequence until it finds a candidate that meets the requirement.

5.4 The Issue of Incompatibility

We have established an appropriate builder algorithm, but face serious questions about its running time.

The problem is that the final candidate is not necessarily a solution. This is because the derived requirements on the inputs may not be sufficient for the output to meet the given requirements. The incompatibility problem discussed earlier is one reason for this. Two candidates for different inputs may both be valid in their own right, but happen to be incompatible with each other. From an algorithmic perspective, this is a constraint satisfaction problem, a problem where a decision needs to be made for each variable (in our case, role), and there are some number of constraints which disallow certain combinations. These problems are NP-complete in general, though fast heuristic solutions exist for many real world examples.

The distressing part of this is that the running time of the builder may be effectively infinite in complex build problems, simply because there are a lot of candidate solutions, and few, or no, valid solutions. In some sense this is reasonable, because with sufficiently complex input there is no other reasonable outcome. However, a software installer is a fundamental part of a working system, and is expected to be robust. It would be far preferable if the system had a guaranteed maximum running time, and could be relied upon even if impossible requests are made.

After a lot of very serious consideration of whether this problem undermined the entire project, the author realised that a general solution was unlikely. Instead, a simple technique from constraint satisfaction problems was used, that of expanding the most constrained variable. When the builder recursively seeks inputs for an indirect provision it then considers the number of candidates for each and iterates over the smallest list. For each candidate, the problem is re-examined, with new requirements if necessary. This should improve the time complexity of the algorithm significantly, but offers no worst case guarantee.

If the prototype is ever developed into a mature system, further analysis of

this algorithm to give some kind of time guarantee on real cases would be a priority. Some potential optimisations are discussed in chapter 9.

5.5 The Issue of Infinity

It is established that complex build problems may generate an enormous solution space, and an effectively infinite running time. However, in certain circumstances, the solution space is actually infinite.

Consider our earlier example of the language L. Recall that X is the source of an L-compiler, written entirely in L, and that Z is a precompiled L-compiler. With just these two entities in the system, there is an infinite progression of candidate L-compilers. If we use the notation A_B to indicate A compiled with B, candidate L-compilers include Z, X_Z, X_{X_Z}, $X_{X_{X_Z}}$, etc.

Formally, this sequence has an initial element Z, and for each element A contains a successor X_A. Direct correspondence to Peano's axioms demonstrates the set is infinite.

If L is well behaved, then the choice of L-compiler does not affect the behaviour of the compiled program, and thus X_{X_Z} and $X_{X_{X_Z}}$ are identical, as they were both effectively compiled by X. However, this is a difficult assertion to convey to the system. Thus, although the possibilities are not really infinite, the prototype must deal with an infinite list of candidates.

There are two reasons why this is a difficulty. The first is that if the builder attempts to completely iterate over this list the task will take forever. The second is that just considering this list, or trying to deduce its size (see previous section), might take forever.

Fortunately, this problem can be satisfactorily solved. The solution to the first issue is to limit the number of recursive reexaminations of the one role allowed on any particular expansion. This limit can be set quite low, and in the

prototype is 4. Semantically, this means that the prototype does not believe any options past $X_{X_{X_Z}}$ are worth investigating.

Secondly, to ensure that a preinvestigation of the solution space does not run forever, it must be ensured that the system acts lazily on a cyclic build problem. That is, the system ought to postpone evaluation of these potentially infinite candidate sequences until required.

This objective is accomplished with the same mechanism as the previous system. The builder tracks the revisiting of roles in any particular expansion. On any second visit, the normal expansion of candidates is blocked, and a substitute list of candidates is supplied which reports its length as infinite and which lazily suppresses evaluation until iteration begins.

Needless to say, through incompatibility or failure to meet requirements the true list of candidates might turn out to be finite. Fortunately, the reported length is used only by the least constrained heuristic introduced above. This means that the discrepancy will at worst reduce the efficiency of the system.

5.6 The Issue of Dependency Attributes

Recall that dependency attributes (see page 33) are attributes which specify that the entity requires additional resources either when it is installed or when it is used.

The use of the relevant entity as a candidate at any point in a build problem requires that additional entities be found to support it. This means that a number of additional parallel build problems must be solved.

There are two different cases here. In an immediate dependency, an input to an indirect provision depends on an additional input. In a permanent dependency, a final product depends on additional supporting software which is needed not just once, but permanently.

Structurally a dependency attribute consists of three things. The first is the required role of the additional entity. The second is an appropriate attribute requirement on the additional entity. The last is a definition of exactly how this supporting entity must be used.

In the case of an immediate dependency, this final definition determines how the additional input is passed to the indirect function. In the prototype, each input is identified by a key, and thus this information is such a key. In the permanent case, the definition determines how or where the additional entity must be installed.

In the prototype development only the immediate dependency case has been addressed. The permanent case is very similar as far as the builder algorithm is concerned. However, the installed supporting entities ought to be tracked, which requires an entity use database (see page 76).

Fortunately dependency attributes can be implemented without an immense change to the builder algorithm. Recall that, having selected a most constrained input, the builder iterates over candidates for that input, and for each candidate recursively reexamines the problem with that candidate selected. If the candidate selected has an attribute which describes a dependency on an additional input at this step, then the recursive reexamination is done with those additional inputs.

There is, of course, a complexity. Recall the dependency attribute must specify how the additional entity is to be used, and in this case, which input to the indirect provision is supplied. It is possible that a candidate has already been selected for that input, presumably because another input required it. It is easy to check that the candidate meets the given attribute requirement. However, it is difficult to define whether the candidate meets a given role.

An example may help to clarify. Suppose that two input roles to the linker,

A and B, are implemented by entities A_1 and B_1 which both additionally require a library. Suppose A is expanded first, A_1 is selected, and it requires the `curses` library. Suppose that this is next expanded, and a candidate library C is selected. When B is expanded, B_1 is selected, and it is discovered that B_1 requires the `ncurses` library, it is difficult to determine whether C is in fact an `ncurses` implementation. This issue is discussed further in the section "Extension versus Compatibility" (see page 70).

5.7 The Issue of Inherited Complexity

It has been established so far that the builder algorithm will struggle to deal with very complex situations, such as when certain entities can be involved in their own construction, when an entity is constructed from a number of input roles with many providers, particularly when a large part of these spaces may fail to meet attribute requirements.

It could be argued that this complexity is acceptable when such complex problems are introduced. However, the problem is far more pervasive.

Recall our earlier example, with W the source of some application, and the infinite progression of L-compilers Z, X_Z, X_{X_Z}, etc. Let *App* be the role of the application, and $WRole$ be the role uniquely possessed by the source file W. Suppose that the indirect provision which details the use of W supplies the function $f : (WRole, LCompiler) \mapsto App$. According to the rules of indirect provision, the candidate space of *App* now includes another infinite progression, $f(W, Z)$, $f(W, X_Z)$, $f(W, X_{X_Z})$, etc.

This is a serious problem. The candidate space is large or infinite not only in certain specific instances where complex issues exist, but any other build problem which is linked to these instances inherits the problem.

5.8 Summary of Final Builder Algorithm

The previous sections have motivated and described extensive modifications to the builder algorithm. This section summarises exactly what the final builder algorithm is.

The function of the builder algorithm is to generate a list of candidate implementations for a particular role which meet an attribute requirement.

Firstly, the builder iterates through the provisions for the given role. It composes a list of candidates from smaller lists resulting from those provisions.

Examining a direct provision produces either a singleton list or an empty list depending on whether the provided entity meets the requirement.

Examining a provision by extension recursively reexamines the entire problem in the extending role.

Examining an indirect provision causes further analysis. The builder determines a requirement for each input. This is normally the same requirement as the builder started with. The active entity is an exception, being required to impart attributes that meet the original requirement and also itself be compatible with the local system.

The builder then recursively examines new problems for each of these inputs, using the input roles given by the indirect provision and the determined requirements. It attempts to gather an approximate size for each of these inputs. The outcome of these calculations might be infinite or zero if the the recursive reexamination is lazily suspended or quashed.

The builder then selects whichever of these inputs is most constrained. It then iterates over the candidates for that input. For each candidate, it recursively reexamines the possibilities for the indirect provision with that candidate selected. If that candidate has a requirement attribute which must be resolved at this stage, then the necessary additional input role and requirement is intro-

duced into the problem.

In this manner, the builder reduces a problem with n unknown inputs to some number of problems with $n - 1$ unknown inputs, and continues until it is examining a problem with 0 unknown inputs, that is, an indirect provision where all inputs are selected. In this case, it establishes either a singleton list or an empty list depending on whether the entity that would be constructed meets the attribute requirement.

With sufficient recursive reexamination the builder can then find the candidate list for any given build problem.

5.9 Operations

Aside from the builder algorithm, the major implementation issue was the implementation of the operations.

Recall that an indirect provision is associated with a function. Evaluating this function runs the relevant program with appropriate arguments and settings. Furthermore, it is also possible to deduce enough information about the output ahead of time to determine whether it meets certain requirements.

Needless to say, this function has to be implemented, including a mechanism for converting the set of inputs into appropriate program invocation (usually command line arguments), and also a mechanism for deducing what attributes the output will have.

In the development of the prototype this functionality was included in the operations. Operations are a subset of entities that have active functionality. Usually this means they wrap an executable file, and the additional code supplied reduces the entity inputs to an appropriate set of command line arguments.

Specifying operations is a difficult job. It requires understanding the functionality of the binary involved fairly well, and understanding of exactly how to

produce appropriate command line arguments for particular inputs.

Chapter 6

Achievements of Prototype Development

As mentioned in the introduction, the objective of this project is not only to design the languages and algorithms needed by the proposed system, but also to demonstrate that the approach is feasible with a prototype.

The previous chapter has discussed the various difficulties of implementing the prototype, and the complexities that had to be overcome. This chapter discusses the achievements of the prototype development, and in particular looks at the degree to which it has been demonstrated that the approach is feasible.

6.1 Implementation of Core Features

In the section "On the Nature of the Proposed System", four core features were set out which the system had to provide in order to address the original objectives.

This section addresses each of these features, and discusses the degree to

which they have been implemented in the prototype.

6.1.1 Multiple Versions and Implementations

This issue is directly addressed by the provision language. This language allows, at the most fundamental level, for the one role to be provided by multiple versions of the same software, and by multiple different implementations.

The prototype entirely implements the provision language.

Thus the project has proven that support for multiple versions can be feasibly implemented using the proposed approach.

6.1.2 Dependencies

The design of the system addresses dependencies on additional software through dependency attributes. Dependency attributes are divided into immediate and permanent dependencies (see page 33). Of these, only immediate dependencies have been implemented in the prototype. The problems involved in implementing these dependencies are discussed in "The Issue of Dependency Attributes"(see page 55).

Thus the project has proven that dependencies can be implemented in at least some contexts, and there is reasonable grounds to believe that dependencies can be fully implemented.

6.1.3 Conflicts

The design of the system addresses conflicts through compatibility attributes. The prototype can implement these attributes easily. However, there is a minor problem. Handling compatibility attributes correctly requires also handling attribute requirements that protect against incompatibility cases. Implementing these requirements via the simplest means is trivial. However, there is a poten-

tial optimisation for the builder algorithm which requires a different approach (see page 77).

Committing to a suboptimal approach was undesirable, and implementing the optimisation (and other comparable ones) in the available time was impossible. As a result, correct implementation of compatibility requirements was never done.

What this effectively means is that the prototype is capable of correctly calculating that certain builds will raise an incompatible outcome, but there is no way to ask it to ensure that this is not the case.

It ought to be emphasised that implementing this inefficiently would not be difficult at all.

Thus, although substantial evidence exists that this feature can be feasibly implemented, this project has not yet proven it.

6.1.4 Variety of Transformations

The prototype includes a mechanism for specifying all manner of active programs, referred to as operations. A number of operations have been specified so far, including the C compiler, assembler and linker. However, these specification is incomplete, in the sense that many of the features and imparted attributes of these operations have been omitted.

However, at this point, it seems clear that there is no obstacle to completely specifying these and other operations, other than time available and the need for in depth understanding of the behaviour of the program being specified.

Thus, it is proven that this aspect of the system is feasible to implement.

6.2 Fulfillment of Additional Advanced Features

At the start of this project, in the section "Aims and Potential Advantages" (page 12), four original aims were set out.

The first two aims were to develop appropriate languages for the project, and to develop a unified system, including developing a prototype. These investigations have been well addressed in the previous chapters.

The third aim of this project was to demonstrate the effectiveness of the approach under investigation by having the prototype perform tasks that are difficult or impossible for comparable systems. A review of these goals is interesting in discerning the degree to which the proposed benefits of the system have been delivered by the prototype.

6.2.1 Tracking Multiple Versions of Compilers

Support for multiple versions of compilers was guaranteed by design. The design of the languages supported multiple versions of any role from the beginning.

However, the original ambition was not just to track multiple compilers and their varying attributes, but also to allow rebuilding of compiled programs on a change to the compiler.

Unfortunately, rebuilding was sidelined fairly early in the work. Since a variety of attribute information is already kept for derived entities, also including historical information is no difficulty.

Some support for this historical bookkeeping, and rebuilding based on it, is in fact present in the prototype. However, rebuilding was sidelined for a reason, namely that better bookkeeping awaits the implementation of an entity use database (see page 76).

Thus this objective was partially implemented, the other part hinging on a subproject that was sidelined.

6.2.2 Using Legacy Compilers

The problem that is addressed here is essentially a compatibility problem. Certain software is incompatible with the approach used in other software. Because of the design of the system, this incompatibility with compilers can be tracked as easily as incompatibility with anything else.

Unfortunately, as mentioned above, compatibility requirements were not implemented in the prototype, although compatibility attributes were. Thus this objective was not entirely met by the prototype, though once again it must be emphasised that it was almost done.

6.2.3 Cross Compiling

One of the most significant potential advantages of the system was the capacity to correctly deal with a cross compiler. Using the attribute language, a cross compiler is simply a compiler whose system compatibility attributes are different to its imparted system compatibility attributes.

The prototype is fully capable of handling these attributes. Furthermore, the prototype correctly requires that all executables it uses in construction must be compatible with the local system. It is also possible to explicitly request a particular attribute requirement for the output, thus make a request for a cross compile.

Support is still fairly unpolished. The system does not default to requiring built entities to be compatible with the local system.

However, this is quite significant progress. The prototype is in this aspect capable of performing a task which its counterparts simply cannot.

6.2.4 Deducing Large Scale Dependencies

One of the interesting possibilities was to deduce the properties of large software objects from those of their constituent parts. Of particular interest is the possibility of deducing the dependencies of a large executable, or of an entire software package, from those of its constituent parts.

The attribute language has made this a possibility, and the prototype can easily trace a dependency from a source file or header through to an executable.

However, the system does not as yet have any concept of a package having summarized dependencies. Although this is a good idea, and would probably be necessary for the system to be used for deployment management, it was not of first priority.

Thus this particular ambition was not fully achieved, although much of the necessary functionality exists.

Chapter 7

Discussion

This chapter discusses the achievements of the project mentioned in the previous chapter, and some of the general difficulties that were encountered.

7.1 Analysing the Achievements of the Project

As this particular project draws to a close it is clear that although a fair amount has been accomplished through the project, the prototype is far from a complete system. As the previous chapter demonstrated, many of the original ambitions of the project were not met, even though in nearly every case the conceptual system was sufficient and the prototype implemented most of the key functionality.

The reasons that the project did not get further are discussed in some detail below. However, what needs to be made clear is that a great deal of progress was made. Although the prototype itself is incomplete, all of the foundations have been laid. This includes not only the conceptual work in the language and the builder algorithm, but also an implementation of most of these conceptual structures and practical analysis of their weaknesses. In addition, substantial

consideration has gone into enhancements that will complete the system. These will be discussed in chapter 9.

Thus, although the prototype is not a complete system now, it serves as a substantial proof of concept for this approach. Although a large scale feasibility study awaits a more complete implementation, most of the difficult issues that need to be addressed in such a complete implementation have already been considered.

7.2 Difficulties and Obstacles

A number of particular obstacles hindered the completion of the system.

7.2.1 The Problem of Generalisation

The original ambition of a project was nothing less than to combine two complex systems, producing an enormously generalised system. Generalised approaches such as this can be very useful, but are more often extremely complicated.

In this case, the difficulties arising in the builder algorithm, those of multiplicity, incompatibility, dependencies and inherited complexity, stem directly from the multiplicity, incompatibility, dependencies and transformations inherent in the system by its very design.

Taken from this perspective the objective of the thesis has been to demonstrate that these different relationships can all be managed at once without the system becoming impossibly convoluted. In this sense, the thesis is a success. As expected, the builder algorithm, which had to make sense of all this information, became quite complicated. However, at the end of the day, the builder algorithm remained an elementary deterministic algorithm, if one with potentially large time complexity.

It was mentioned earlier that the multiplicity and incompatibility in the

system already introduce a constraint satisfaction problem. Since the system is therefore solving something an NP-complete problem, it is impossible that the worst case time complexity is polynomial.

7.2.2 Establishing a Framework

It must be understood that the system that has been developed here is in essence a framework for solving configuration and build problems. As in developing any framework, the difficulty is that all decisions are made in a conceptual vacuum. It being impossible to know ahead of time what assumptions may be made about any other part of the system and what assumptions might be required, these decisions have to be made without a clear idea what design requirements are present. As a result, a lot of time is wasted simply trying to see forward to potential problems with a number of design decisions, and certain decisions were wrong, as will be discussed below.

These problems were particularly acute because the system designed here does not greatly resemble any other. Although in hindsight, substantial parts of the system, particularly the provision language, do resemble others, this was not an initial design decision.

The particular impact of this issue on the project was that the rapid rate of development anticipated by the earlier planning was not attainable. Too much time was spent considering the potential implications of minor decisions.

7.2.3 Lack of Accessible Examples

Development of a build system needs a number of example build problems on which to test it, conceptually and in prototype. Likewise, development of a configuration system needs some examples, and in fact more complex examples.

Unfortunately for this project, the examples available were fairly neatly par-

titioned into those that were not sufficiently complex to need the advanced functionality of the system, and those that were so complex that specifying their build structures in the available time was absolutely impossible.

7.2.4 Extension versus Compatibility

One of the poor decisions that was made in this project was to include both compatibility attributes and provision by extension. Semantically, each of these allow additional assumptions to be made about certain entities.

In the case of provision by extension, an explicit additional role is created in which the entities with greater capacity reside. In the case of a compatibility attribute, the entities with the greater capacity are explicitly declared as having greater capacity.

There are, of course, advantages to using either approach.

The compatibility attribute approach is fine for direct provisions, however for an indirectly provided entity to have an exceptional attribute, all its parents must, which may be unrealistic if some are utility libraries. However, the advantage of compatibility attributes is that these entities can always be checked against a requirement.

By comparison, the provision by extension approach is easy to implement, but there is no way to check whether entities from the general role provide the extended role. As an additional benefit, queries against the extended role do not need to consider any provisions of the general role. By comparison, searching for implementations of the general role which meet a compatibility requirement involves a complete search filtered by this requirement.

As a result of these considerations, both systems were carried to implementation. In hindsight, the fast search property of the provision by extension could be provided by optimisations that are mentioned in chapter 9. If these opti-

misations had been considered from the start, a different approach might have been selected.

Chapter 8

Conclusions

The original objective of this thesis was to address the question of whether build systems and configuration systems could be unified on a practical level.

This thesis has well addressed the conceptual issues involved with this problem. The attribute and provision languages and builder algorithm which form the core of such a system have been designed. The capabilities of this approach have been proven with a prototype. Furthermore, key practical issues in developing such a system have been isolated, and a number of approaches to dealing with them considered. Finally, the original objectives and the selected approach have been validated through proof that the prototype is capable of handling difficult problems which other similar systems struggle with, such as multiple compilers with different attributes.

Chapter 9

Possibilities for Further Work

The body of this work has focused on designing the attribute and provision languages and developing the builder algorithm for the prototype. The capacity of these techniques has been investigated, but the feasibility of using them in practice remains in doubt.

Thus, the principle opportunity for further work is in investigating the practical advantages and limitations of using this approach. There are a number of specific smaller projects that would be involved in such a study.

9.1 Specification of More Examples

In this work, examples have been developed mainly for the purposes of testing the prototype and of demonstrating the capabilities of the languages. For this reason, these examples have all been simple or contrived.

Simple examples were useful in prototype development, as the time needed

to develop more extensive examples and to extend the prototype to support sufficient features was not available.

The problem with using simple examples rather than real world case studies is that the evaluation possible so far is purely academic. To investigate the practicalities of the system, and in particular its capacity to deal with large scale issues, an extensive library of case studies should be developed.

9.2 Specification of More Operations

On a closely related note, only a small number of applications were specified in the prototype. This was entirely due to time constraints, and closely linked with the fact that only a small number of simple examples were developed.

In order to specify a more expansive set of examples, a number of additional operations will also have to be specified. In addition, programs like gcc that have been specified are missing most of their features. To provide features needed by more complex case studies, these operations will have to be specified much more completely.

9.3 Interoperation with Other Build Systems

Respecifying any significant chunk of the world's codebase using an alternative build system is an enormous task. Most of this software is already specified using one or another existing build system. Thus a mechanism for automatically performing all or part of the conversion from a mainstream specification language to the ones developed here would be of enormous value.

Such projects tend to be difficult. In particular, the build languages developed here require a fair amount of attribute information that simply is not present in other systems. Thus it is unclear whether this automatic conversion

is possible at all. In any case, it remains an interesting avenue for exploration.

9.4 Scientific Analysis of the Builder Algorithm

The major concern arising out of the prototype implementation was that the builder algorithm might have exponential complexity on sufficiently difficult build problems.

This issue has already been investigated, considered and discussed in this project, but only in an abstract manner, with purely hypothetical guesses at the real time complexity. What is needed is some form of scientific analysis. Using the library of case studies, a suite of appropriate tests should be developed. This would allow a study of the builder algorithm with measurable performance data.

It seems likely that this study would determine certain situations in which the builder algorithm is particularly inefficient. Hopefully this would lead to further optimisation, or to a study of how to avoid these situations in practice.

9.5 Robustness

This thesis has focused mainly on the functionality of the languages and builder when all inputs have been correct. No effort has been made to make the prototype deal robustly with erroneous or incomplete input from the user or from poorly specified packages and entities. To develop the system for wide scale application it is of great importance to ensure the system can detect and deal with errors in specification without forcing the user to understand its internals.

This would require at least a well defined error reporting system. Ideally, however, a failed operation could interpret the error message from the operation and make a reasonable guess as to what is missing or incorrect in the specifica-

tion. If this feature could be developed it would be of great value to the system's dependability.

9.6 Development of an Entity Use Database

Although the languages were designed with a variety of applications in mind, the prototype development focused on the build system side of things. In particular, there is no facility yet developed for installing a given program into appropriate parts of the file system.

Such installation is core functionality to a configuration system, and is thus a significant omission. This omission perhaps reflects the fact that the project has been interested primarily in the design and behaviour of the system, and has approached the task of implementation as a second priority.

Implementing installation would not be especially difficult. The dependency attributes that already exist can be easily generalised to include dependencies on installed software. However, if objects are to be linked or moved into the filesystem because they are needed by others, then some form of tracking is needed. In particular, installed software ought to be cleaned up once it is no longer needed. This means that some sort of database ought to be kept which stores information on which entities are needed for various reasons.

This database is useful for more than just cleaning up unused entities for the configuration system. It can also be used for tracking updates. Conventional build systems work backwards to perform updates, starting with a given target and recursively checking that its parents are built and up to date. This can be enormously inefficient when small changes are made to large systems. A better method, and one which is much more practical on client machines that are receiving changes to sources rather than making them, is to work forward. Starting with the changed sources, the program calculates what their derivatives

were, and can work forward rebuilding things until all derivatives are rebuilt.

The calculation of the derivatives would come straight from the entity use database. This is done by including in the use database record for entity A not only those dependent entities which require A, but also dependent entities which were built from A.

9.7 Optimisation for Compatibility Attributes

Recall that compatibility attributes are used to model various possible conflicting assumptions. The significant unique feature about these attributes is that when conflicting assumptions are combined the result is a an identifiable 'bottom' attribute.

In some situations, such as system compatibility attributes, any given build problem will include an explicit requirement for compatibility with a given system. However, in many situations the user does not greatly care which assumptions are made, only that there is no incompatibility in the final outcome.

The obvious approach to this problem is to set a requirement which is met by any attributes apart from this 'bottom' attribute. The general problem with this approach has already been highlighted in "The Issue of Incompatibility" (see page 52). In summary, the problem is that the requirement on the inputs does not prevent the output being invalid, thus an enormous number of invalid candidates may have to be considered.

A different approach, and one which could greatly improve performance, is to rearrange the requirement. Instead, consider the requirement as a requirement that the given entity must have an attribute value greater than or equal to some element of a set of admissible values.

The advantage is that this requirement can be manipulated more intelligently. For instance, it can be unified with another requirement intelligently,

though keeping the set sizes small is complex.

In addition, this requirement can allow a more intelligent handling of compatibility in an indirect provision. Suppose the builder is seeking to use an indirect provision, and has such an attribute requirement. When a candidate input is selected, the requirement can be contracted to only those elements of the set of values which the candidate meets.

9.8 Caching of Candidate Searches

One of the serious concerns about the system so far is that the builder algorithm's search for candidates for a particular role may require a significant amount of time to yield only a few candidates. One straightforward solution to this problem is to cache the results of these searches. This could in particular solve the problem of inherited complexity (see page 57).

The complication in implementing this approach is that searches are performed on both a role and an attribute requirement. Thus a great number of fairly similar searches can be performed which use the one role, but varying attribute requirements. To make this approach work, some mechanism for reducing an attribute requirement down to those parts which most significantly constrained the solution space is necessary.

These reduced attributes can be used in order to ensure a reasonable number of cache hits. They could also be extremely useful in error reporting, as they could be used to define precisely which part of an extensive requirement prevented any candidates being found.

9.9 Extensions to the Application Domain

The body of this document has discussed an alternative approach to software construction and deployment. This approach unifies the build and configuration systems. Many of the advantages of this approach derive from nothing more than communication between the build and configuration levels. Compared to conventional approaches, where the configuration system can only affect the build process by rearranging the filesystem, this yields substantial benefits. The final proposal here is that additional benefits can derive from extending this level of communication to arbitrary applications.

Consider the problem of configuring the system as established earlier. For each object in the system we may pick up a series of assumptions it makes about what will be where in the filesystem. This can lead to a number of problems. The problem of dealing with legacy compilers discussed earlier (see page 13) is just a single case of the general problem of trying to give two different things the same name in the filesystem. The simplest solution to this problem is to avoid filesystem names entirely and instead locate the resources provided by software by querying the configuration system.

Fixed installation locations were undesirable anyway. They rob users and administrators of any control of the layout of the system. Specifically, they perpetuate the inane tradition that almost everything in the system should be located somewhere in `/usr`, unless it was installed by a user. Perhaps more importantly, fixed file systems tend to lead to the development of software that can only be used on similar filesystems. This is most common in the free software community, where large quantities of software is completely dependent on the UNIX approach.

Recall a key turning point in the history of the HTTP protocol, where it was realised that rather than using a static archive, a server can respond to a

request by dynamically generating the resource requested. Our configuration system too could provide resources generated dynamically. This could allow resources supplied in unexpected manners, such as files being provided which are generated on demand, or libraries being provided by interpreted languages or remote procedure calling.

The most straightforward application of dynamic supply is "lazy" installation of software, that is, installation of software on demand without requiring any special action by the software making the demand. For instance, if a typesetting package like latex is requesting font files, then the configuration system can suspend the request, consult the user, and install the relevant files on the fly. This is a huge improvement over current mechanisms where the program simply generates a "file not found" error, and the user is left to determine how to fix the problem.

All of these possibilities are very interesting, and motivate the development of a flexible application interface to the configuration system. Such an interface should allow software to request data files, processes and libraries by appealing to the configuration system's understanding of roles.

Use of this library by any significant amount of software could lead to a number of advantages. We can successfully deal with multiple versions of arbitrary software roles, provide software on any system of any layout, and allow more interesting possibilities in terms of what programs can fulfil software roles.

Bibliography

[1] Eelco Dolstra. Integrating software construction and software deployment. In Bernhard Westfechtel, editor, *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, Portland, Oregon, USA, May 2003. Springer-Verlag.

[2] S.I. Feldman. Make - a program for maintaining computer programs. Computing Science Technical Report 57, Bell Laboratories, 1977.

[3] P.A. Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc.*, 19(1):14–25, 1998.

[4] *GNU Make*. http://www.gnu.org/software/make/.

[5] *Jam*. http://www.perforce.com/jam/jam.html.

[6] *Make Module*. http://www.livinglogic.de/Python/make/.

[7] *Rake*. http://rake.rubyforge.org/.

[8] *Rubuild*. http://www.bct-portal.com/opensource/ruby/rubuild/index.

[9] *Rule Based Command Execution*. http://bras.berlios.de/.

[10] *Apache Ant*. http://ant.apache.org/.

[11] *CONS: A Make Replacement*. http://www.dsmit.com/cons/.

[12] *SCons: A Software Construction Tool*. http://www.scons.org/.

[13] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The vesta approach to software configuration management. Research Report 168, Compaq Systems Research Center, 2001.

[14] Allan Heydon, Roy Levin, Timothy Mann, and Yuan Yu. The vesta software configuration management system. Research Report 177, Compaq Systems Research Center, 2002.

[15] *Red Hat Package Management System*. http://www.rpm.org/.

[16] *Debian GNU/Linux Operating System*. http://www.debian.org/.

[17] *APT for RPM Based Distributions*. http://apt4rpm.sourceforge.net/.

[18] *GNUpdate*. http://www.gnupdate.org/.

[19] *Fink Project*. http://fink.sourceforge.net/.

[20] *The FreeBSD Project*. http://www.freebsd.org/.

[21] *OpenBSD*. http://www.openbsd.org/.

[22] *The NetBSD Project*. http://www.netbsd.org/.

[23] *Gentoo Linux Distribution*. http://www.gentoo.org/.