University of Alberta

Build System Issues in Multilanguage Software

by

Andrew Neitsch

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

©Andrew Neitsch
Fall 2012
Edmonton, Alberta

Abstract

Software build systems are often viewed as a solved problem by software engineers, since there are many mature tools and techniques for implementing them. However, existing tools and techniques focus on building single-language software and do not necessarily scale to building multilanguage software. An exploratory qualitative study of five open-source multilanguage software packages finds significant issues in getting these packages to build at all. The key finding of this thesis is that there are commonalities among build problems that could be systematically addressed. A comparative analysis of the five case studies produces a set of build patterns and anti-patterns, some of which could also apply to single-language software. One such finding is that multilanguage packages may avoid certain build problems by supporting build-free extension. As well, there is evidence that abstractions from the application and implementation domains may “leak” into the build system, with both positive and negative effects.

Acknowledgements

Thank you to Ken Wong, Mike Godfrey, and Abram Hindle; my family and friends; the anonymous reviewers of publications based on this work; and the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Build systems turn source code into executable programs by orchestrating the execution of compilers, code generators, and other tools. They are also responsible for turning incrementally updated source code into incrementally updated executable programs. They operate by taking as input a specification of which artifacts are produced from which other artifacts by which tools, and executing the tools necessary to bring all artifacts up-to-date. The classic example of a build system is a Makefile interpreted by Make [19].

Build systems are commonly viewed as a "solved problem" by software engineers, because there are many mature and well-known tools and techniques for creating build systems. Some are fully automatic. For example, when developing Java source code using the Eclipse Integrated Development Environment (IDE) available from eclipse.org, there is a "Build automatically" option which is enabled by default. With no additional setup, this option automatically gives near-instantaneous builds for Java source code. Many other build tools have been developed with advances in terms of convenience, correctness, or speed [60]. For example, automatic dependency extractors help to infer build system specifications from source code, partially automating a tedious and error-prone manual process; use of checksums to detect changed artifacts avoids some edge cases with older timestamp-based checks, such as cases where artifacts are edited while a build is in progress; and cloud computing tools have been developed that automatically parallelize builds to the maximum extent possible [50].

However, most of these advances focus on building software written in a single programming language. What about building software written in multiple programming languages? For example, Eclipse also supports automatic builds of C/C++ software. It is not uncommon for Java software to call C/C++ routines via the Java Native Interface (JNI); this is how Java software is able to access the platform windowing system, for example. However, Eclipse has no support for JNI builds. Developers wishing to use Eclipse for combined Java–C/C++ development via JNI must understand the internals of Eclipse's automatic build system, as well as Java and C++ build tools, in order to hook into Eclipse's auto-generated

Makefiles to automatically update JNI header files when Java classes change [6, p. 10].

This thesis examines build systems for multilanguage software. The term *multilanguage software* is used to refer specifically to software that is written using multiple programming languages, and in which the parts written in different languages are both necessary and interdependent in the implementation of the software. This definition is necessary to distinguish 'interesting' cases where multiple languages are tightly integrated from less-interesting cases where multiple programming languages happen to be used. Not every software package written using multiple programming languages satisfies this definition. For example, a database server written entirely in C is not multilanguage in this sense merely because a Java client library is available, because an optional client library is not a necessary part of the server implementation. Nor would this hypothetical database server be considered multilanguage if its developers used a unit test driver they had written for it in Tcl, or if its build system were written in Ruby, or if the server shipped with a standalone GUI documentation viewer written in Java. In these situations, the part written in the other programming languages is not part of the database server implementation. However, if the database server supported stored procedures written in Java, and the database had, for example, a full-text search feature internally implemented in part as stored procedures written in Java, then the database server would be considered multilanguage, because the Java and C parts would be interdependent and both would be necessary in the implementation.

Multilanguage software has the potential to combine the unique strengths of different programming languages while cancelling out their weaknesses. There are many thousands of extant programming languages, created for many different reasons [55]. Multilanguage software allows leveraging more than just one of them. For example, consider the Emacs editor, in either its XEmacs or GNU Emacs manifestation. It is multilanguage software written in both Lisp and C. Any individual function can be implemented in either language, and changing the implementation language requires a recompilation but no other source code changes. This allows either language to be used throughout, wherever it is most convenient. For example, syntax highlighting is implemented in Lisp, to take advantage of its expressiveness and run-time extensibility. This allows syntax highlighting definitions to be customized at run time. In contrast, the terminal setup and display update routines are written in C to take advantage of its low-level system access capabilities and its speed. However, in order to get this combination, Emacs implements its own Lisp interpreter in C.

Anecdotal experience suggests that building multilanguage may not yet be "solved," and that tools and techniques for building single-language software may not scale to building multilanguage software. In addition to the Eclipse JNI example, consider that a developer wishing to create a Lisp–C application software package using an Emacs-like multilanguage architecture may have to complete the extensive task of writing a Lisp interpreter in C before being able to run a single line of Lisp code of the actual application software. Another researcher, Bram Adams, investigated adding aspect-orientation to C programs, thereby creating a form of multilanguage software. He encountered sufficiently many build system problems and sufficiently little existing research on the subject that his doctoral dissertation focuses on managing interactions between build systems and source code [2]. Multilanguage software may have appealing advantages in the abstract, but these may appear to be outweighed by initial concrete difficulties. Multiple build systems for source code in multiple programming languages must be developed and integrated before being able to develop the actual multilanguage software, and the resulting customized hybrid build system must be maintained as well. Potentially complex build systems may be perceived as a deterrent to creating multilanguage software.

This thesis systematically examines build systems for multilanguage software in order to determine if build systems are problematic for multilanguage software, and if so, what might be done about it.

The specific research questions of this thesis are:

RQ1) What are the major issues in building multilanguage software?

RQ2) How can these build issues be addressed?

RQ3) Why do these build issues occur?

There are three major stages of this qualitative exploratory study. First, packages to be studied are systematically selected from the Ubuntu 9.10 "Karmic Koala" operating system distribution. Then, case studies are conducted on each package to understand the multilanguage and build system architectures of each, and to record observations of both the build problems encountered and the build features that prevent problems. Finally, a comparative analysis of results from the case studies is summarized in a set of *build patterns* and *anti-patterns*.

The key finding of this thesis is that build problems for multilanguage software are error-prone, but there are commonalties among the problems which could be systematically addressed.

1.1 Contributions

The contributions of this thesis are as follows:

- *A procedure for identifying a diverse selection of multilanguage packages*, including discussion of how well it worked and what its limitations are. The selection procedure uses filename patterns to identify the programming languages used in 101GB of source files. This procedure is described in Chapter 3, including details of Ubuntu's package metadata, and the mechanics of extracting and using it. This knowledge could be useful to future researchers dealing with multilanguage packages.

- *Deep exploration of the build systems of five multilanguage software case studies*, as well as exploration of the multilanguage architecture of the case studies themselves. This partly addresses RQ1, and is detailed in Chapter 4. The observed build problems, how to resolve them, and build features that help to avoid problems are described for each package. The logical multilanguage architecture and the build structure of each package are also explained, with corresponding diagrams, such as Figure 4.5 on page 39 and Figure 4.20 on page 62. The notation of the diagrams is based on the work of Godfrey and Tu [68], but the diagrams here use a higher level of abstraction.
- *The finding that four of the five multilanguage case studies have error-prone build systems*, in the sense of requiring manual intervention to get a running development version from source code. The fifth package's build system has no problems building, but does have some problems rebuilding. Since these packages are diverse along several important dimensions and were selected without advance knowledge of how they would build, and since all of them have build or rebuild issues, it is likely that build systems for multilanguage software are, in general, error-prone.
- *Build patterns and anti-patterns* summarize the commonalities among build problems, and how they may be dealt with. These address RQ1 and RQ2, and comprise Chapter 5. The build patterns and anti-patterns are inferred from a comparative analysis of the build problems and features observed in the case studies. Table 5.2 on page 76 links each pattern or anti-pattern to numbered observations in the case studies, and its dual table is Table 5.3 on page 77.
- *Discussion of potential uses and implications of the patterns and anti-patterns*, and how they address the research questions. This discussion comprises Section 5.9 and Chapter 6. Section 6.2 also discusses the phenomenon of abstractions 'leaking' from application and implementation domains into the build system, which may help address RQ3.

Chapter 2 gives necessary background information about build systems and other concepts that are used throughout this thesis. Chapter 3 details the systematic procedure used to select the case study subjects, and includes its own discussion, threats to validity, and related work sections. Chapter 4 gives in-depth case studies of each selected software package. Chapter 5 describes the build patterns and anti-patterns. Chapter 6 discusses the results, their generalizability, and their limitations, along with related work. Chapter 7 concludes this thesis and discusses future work.

A peer-reviewed conference paper based on this research is also in press [48].

Chapter 2

Background

This chapter provides necessary definitions related to build systems that will be used later in this thesis.

Dependencies are relationships among tools and artifacts in a software package. If a tool takes artifact B as input and produces A as output, then A is said to *depend* on both B and the tool. For example, when a compiler produces an object file `foo.o` from a C file `foo.c`, then `foo.o` depends on the compiler and all inputs to the compiler: header files included by `foo.c`, system libraries loaded by the compiler, the compiler's command-line options, *&*c. Build systems use specifications of such dependencies to automatically build project deliverables by running the appropriate tools with the appropriate inputs.

When a project has already been built and some of the input artifacts change, such as when a source code file is modified by a developer, the build system is responsible for rerunning the appropriate tools to bring all outputs up-to-date with respect to their inputs. In most builds, there are many more dependencies that could change than can be checked in a reasonable amount of time, so build systems use shortcuts. For example, many build systems compare file timestamps to check whether input artifacts have changed since output artifacts were generated. However, changed files may appear with old timestamps when restored from backups, and an input file that is changed while a corresponding output file is being produced can cause the output file to be out-of-date, even though the input file has an older timestamp. Many build systems will also omit checking for changes to system header files and compiler versions, even though these are occasionally upgraded. In theory, the key challenge in build system design is to specify and check sufficiently many dependencies that outputs are consistent with inputs, but not to perform so many checks that it would be faster to regenerate artifacts than to check whether they are up-to-date.

Dependency graphs are defined in terms of manufacture graphs, a concept formalized by Borison in the 1980s [10, 11]. A *manufacture graph* is the directed graph induced by the depends relation on the artifacts and tools. It is usually a bipartite graph, where artifacts alternate with tools on each path, and edges exist from each input artifact to each corresponding tool invocation, and from each tool invocation to each corresponding output artifact. The graph is not bipartite when an artifact produced during the build is later used as a tool in the same build. A *dependency graph* is a graph of dependency relations among artifacts that is produced from a manufacture graph by adding an edge from each input artifact to each output artifact that is reachable from it by passing through a single tool node, and then removing all tool nodes.

The specification of all dependencies for use in the dependency graph can often be automated to a large degree through the use of *automatic dependency extraction* also known as *automatic dependency analysis* [60] or *dependency scanning* [38]. Tools implementing automatic dependency extraction are able to infer the dependencies of artifacts from the contents of the artifacts themselves. For example, an automatic dependency extractor for C builds can scan C source code files to determine the header files they include, and automatically specify them as dependencies for the corresponding object files.

Using the concept of a manufacture graph, a software build can be divided into three conceptual phases: configuration, construction, and packaging. *Configuration* is deciding what to build: deciding, based on platform capabilities and developer input, exactly which artifacts to produce from which other artifacts by using which tools. For example, which C compiler to use with which compilation settings, and which optional features to include in the built software, are the types of decisions that can be made during this phase. The output of this phase is a manufacture graph. The *construction* phase takes the manufacture graph generated by the configuration phase and traverses it, calling tools as required to bring all artifacts up-to-date. The output of the construction phase is an executable software artifact. Finally, the *packaging* phase gathers artifacts for distribution. The output of the packaging phase is often a platform-specific installable package, such as a Ubuntu `.deb` file, a Mac `.dmg` file, or a Windows `.msi` file. In actual build systems, there is often some overlap between these conceptual phases; for example, some details of the configuration may be deferred to the construction phase, and the installable package is often an artifact produced by the construction phase. Nevertheless, these phases are useful concepts for discussing build systems.

The requirements for each phase, and common methods of implementing them, vary by programming language:

- Java's write-once-run-anywhere design and absence of conditional compilation results in minimal configuration requirements. Construction is straightforward since all sources can be quickly byte-compiled with a single tool invocation. Packaging is often nothing more than zipping the byte-compiled `.class` files into a JAR archive. Most of this can be handled completely automatically by, for example, Eclipse.

- In contrast, C/C++ software has extensive configuration requirements because of the huge diversity of platforms it may run on. The construction phase is complex, as conditional compilation via the preprocessor and the time-consuming nature of C/C++ compilation place heavy demands on incremental recompilation tools. There are are many operating-system-specific packaging formats.

  One framework for addressing the build needs of C/C++ software is autotools [66, 74]. It consists of a number of related tools, including autoconf and automake. The autoconf tool generates configuration probes as portable shell scripts, since a working command shell is essentially the only commonality among all the operating systems it is designed to work on. The configuration probes determine what platform features and tools are available by attempting to use them and recording whether the attempts succeed. The output is typically recorded as a header file of preprocessor definitions for use with conditional compilation. The automake tool automatically generates a Make-based build system that uses the makedepend automatic dependency extraction tool. Packaging is accomplished by creating an archive file that contains the automatically-generated configuration script.

- Software packages written in scripting languages such as Python and Ruby have simple configuration requirements. Since source code runs on a portable runtime abstraction layer, platform-specific configuration is usually not needed. However, extensive use of third-party libraries by these development communities makes prerequisite library satisfaction an important configuration-time requirement. Construction and packaging involve only direct file copy for some languages or archiving of byte-compiled code for others.

  Build tools for such languages tend to be libraries with declarative APIs. For example, Python package build systems often consist of a file called `setup.py`, containing a single function call that passes metadata to the `distutils` library. The `distutils` library can use the metadata to automatically handle tasks such as installing prerequisites from the web, installing packages locally, building byte-compiled distribution files, and uploading them to the central community Python library server.

As these examples show, build tools are often highly specialized for one language, and may not be applicable to other languages. There are fundamental differences in what the tools are designed for. For example, C builds are usually artifact-based with configuration-phase emphasis on determining platform capabilities, while Python builds are task-based with configuration-phase emphasis on installing prerequisite libraries.

An *artifact-based* build tool is one in which artifacts—usually files—are the primary unit for specifying the build system [33]. One example of such a tool is Make. In contrast, a *task-oriented* build tool is one in which the primary units of specification are tasks. Tasks in this context refer to stages of the build, such as compilation, running unit tests, or creating an installable package. One task-oriented tool is Apache Ant. Consider the difference in how one might add a unit-testing step to a build system implemented using each tool. In Make, one could add a target file `unit-test-report.txt` that depends on individual executable unit test files. In contrast, in Ant one could add a "unittest" target that depends on the "compile" target which represents the act of compiling all source code, rather than any particular file. Another example of an artifact-based build tool is Daniel J. Bernstein's `redo` [8, 31], in which build commands are specified for each artifact, and dependencies are dynamically recorded during builds.

A *build framework* is a collection of tools, techniques, conventions, and/or template files that are used together. Examples could include: Make together with a Makefile framework file and the `move-if-change` script [60]; the autotools suite; or distutils together with the libraries it uses.

An object-oriented build tool is one implemented as an API in a general-purpose programming language, and an *object-oriented build system* is a build system implemented using such tools. Object-oriented build systems are composed of objects that can be subclassed, queried, and mutated. Such build systems can take advantage of reusable third-party libraries of build functionality. Examples of potential build functionality include documentation generation and code quality metric reports. Examples of object-oriented build tools include Rake [20] and SCons [37].

Apache Ant is a popular build tool for Java software that, while it does provide an API in a general-purpose programming language, is not an object-oriented build tool as defined here. The reason is that Ant's primary interface is a custom XML-based programming language that does not allow for dynamic interaction with the Java objects that implement Ant's builds. While Ant does support loading third-party libraries that provide additional build functionality, individual projects cannot dynamically extend or modify their Ant-based build systems from within Ant. Projects wishing to extend or modify Ant-based build systems must build extension modules separately before invoking Ant.

Chapter 3

Selecting multilanguage packages

This chapter describes the systematic process for selecting the case studies in Chapter 4. There are two major issues to address: how to identify multilanguage software, and which of the identified multilanguage software packages to select. Identifying multilanguage software is accomplished using a combination of automatic and manual procedures on the software packages in the Ubuntu 9.10 “Karmic Koala” distribution. Packages that may be multilanguage are identified by automatically examining filenames. However, having source code files in multiple programming languages does not necessarily imply that that a software package is multilanguage in the sense of having multiple interdependent implementation languages. To determine whether identified candidate packages are actually multilanguage, manual investigation is performed. Finally, a subset of the multilanguage packages are selected as case studies using random selection.

In software engineering studies in general, it is common not to explain how examined cases are selected. Page space in publications is often severely limited, and is better spent on the results found by examining the cases than on the somewhat tangential topic of how those cases were selected in the first place. Additionally, many studies use familiar open-source projects such as Linux, Quake, Eclipse, and Mozilla as benchmarks. Studying benchmark cases means that other researchers understand results more easily because they are already familiar with the cases. More importantly, researchers are able to compare results [59]. However, there is potential for bias in always looking at the same cases, and most of the well-known benchmark cases are not multilanguage. There is also potential for bias in selecting cases that researchers conducting the study have already had experiences with. For these reasons, this thesis does not use benchmark cases, or cases that the author liked or disliked, but rather random selection.

The packages are selected from among the Ubuntu 9.10 distribution. Ubuntu is one of the most popular Linux distributions. It has a huge number of software packages, representing much of the open-source software ecosystem. Ubuntu has more than 29 000 installable binary packages built from 16 153 source packages. The exact number of binary packages depends on the platform, as there are Intel-, Sparc-, ARM-, and PowerPC-specific packages. There are more binary packages than source packages because artifacts built from source packages are split into multiple binary packages for efficiency reasons—end-users can save download time and disk space by omitting parts of software that they do not need.

In addition to a large number of available packages, one benefit of using Ubuntu, as described by González-Barahona et al. [29], is that all of the package metadata is available on the web in an easily-parsable plain text format. The metadata can be used for tasks such as downloading source code, understanding the relationship between source and binary packages, and determining software dependencies. To demonstrate how these tasks can be performed using the metadata excerpts in Figure 3.1 and Figure 3.2, consider the following entries from the metadata files:

```
Package: openssl
Binary:
 openssl, openssl-doc, libssl0.9.8, libcrypto0.9.8-udeb, libssl-dev,
 libssl0.9.8-dbg
Version: 0.9.8g-16ubuntu3
Priority: important
Section: utils
Maintainer: Ubuntu Core Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer:
 Debian OpenSSL Team <pkg-openssl-devel@lists.alioth.debian.org>
Build-Depends:
 debhelper (>= 5.0.22), binutils (>= 2.14.90.0.7), zlib1g-dev, m4, bc
Architecture: any
Standards-Version: 3.8.0
Format: 1.0
Directory: pool/main/o/openssl
Files:
 1ff3251e3282265da35c7e1374350252 1429 openssl_0.9.8g-16ubuntu3.dsc
 acf70a16359bf3658bdfb74bda1c4419 3354792 openssl_0.9.8g.orig.tar.gz
 e17725c55943ef402de033789c201ab2 61628 openssl_0.9.8g-16ubuntu3.diff.gz
Uploaders:
 Christoph Martin <christoph.martin@uni-mainz.de>,
 Kurt Roeckx <kurt@roeckx.be>
Vcs-Browser: http://svn.debian.org/wsvn/pkg-openssl/openssl
Vcs-Svn: svn://svn.debian.org/pkg-openssl/openssl/
Checksums-Sha1:
 4e9c5ced46...bde5c15da80fa1 3354792 openssl_0.9.8g.orig.tar.gz
 9358d09d50...9edb6c6d22538e 61628 openssl_0.9.8g-16ubuntu3.diff.gz
Checksums-Sha256:
 0e26886845d...aa5f7ee901cf0 3354792 openssl_0.9.8g.orig.tar.gz
 2be42444fa9...e8444707ef272 61628 openssl_0.9.8g-16ubuntu3.diff.gz
```

Figure 3.1: Ubuntu 9.10 `Sources` excerpt for `openssl` source package

```
Package: openssl
Priority: important
Section: utils
Installed-Size: 836
Maintainer:
 Ubuntu Core Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer:
 Debian OpenSSL Team <pkg-openssl-devel@lists.alioth.debian.org>
Architecture: amd64
Version: 0.9.8g-16ubuntu3
Depends: libc6 (>= 2.7), libssl0.9.8 (>= 0.9.8f-5), zlib1g (>= 1:1.1.4)
Suggests: ca-certificates, openssl-doc
Conflicts: ssleay (<< 0.9.2b)
Filename: pool/main/o/openssl/openssl_0.9.8g-16ubuntu3_amd64.deb
Size: 402260
MD5sum: 29b88b1a0a1898b5d7c41397407ee2cb
SHA1: 0c13848bf10d4a2baa54279ae22ac7372601c33b
SHA256: 1ec6f18bc8d66f9027aeb8e4c5bcb63d6b02efbce0d551a53cd95eb0b1254e02
Description: Secure Socket Layer (SSL) binary and related cryptographic tools
 This package contains the openssl binary and related tools.
 .
 It is part of the OpenSSL implementation of SSL.
 .
 You need it to perform certain cryptographic actions like:
  -  Creation of RSA, DH and DSA key parameters;
  -  Creation of X.509 certificates, CSRs and CRLs;
  -  Calculation of message digests;
  -  Encryption and decryption with ciphers;
  -  SSL/TLS client and server tests;
  -  Handling of S/MIME signed or encrypted mail.
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Build-Essential: yes
Origin: Ubuntu
Task: minimal
```

Figure 3.2: Ubuntu 9.10 `Packages` excerpt for `openssl` binary package

Package	Description
`openssl`	`openssl` command-line program
`openssl-doc`	documentation
`libssl0.9.8`	shared library containing `openssl` functionality
`libcrypto0.9.8-udeb`	documentation
`libssl-dev`	header files
`libssl-0.9.8dbg`	debug symbols

Table 3.1: Binary packages built from the `openssl` source package

1. The `Directory` and `Files` entries can be concatenated with the URL of a Ubuntu distribution site to yield a download link for the original vendor source code archive.[1]

2. The `Binary` entry in the source package metadata lists the six binary packages that are produced from the source package; see Table 3.1.

3. The `Build-Depends` and `Depends` entries list the other packages that are required to build and run the software.

Dependency metadata may appear useful for identifying multilanguage packages, but is insufficient. Since packages are required to list their build- and run-time dependencies, it appears that one could determine which languages are used in packages by looking for programming-language-specific compilers and runtime libraries in `Depends` entries. But there are two problems with this: (1) the default set of required packages is too large, and (2) packages are not necessarily implemented in the same languages as the packages they depend on.

1. Note the `Build-Essential: yes` entry for the `openssl` binary. This flags the binary package as a prerequisite for building any package. There is also a related `Essential` flag for designating a package as required on every Ubuntu system. There are 74 such `Essential` or `Build-Essential` packages in Ubuntu, including `gcc`, `g++`, `perl`, and `python`. Since these are automatic prerequisites, they are not listed in any dependency metadata.

2. For efficiency reasons, dependency specifications are automatically transitively reduced. That is, if package A depends on B and C, but B also depends on C, then only the dependency on B is recorded in A's metadata. Because software package dependency satisfaction is a transitive relation, this suffices for the Ubuntu package management system, which is the intended client of dependency metadata. However, this precludes using dependency specifications to determine language use, because software need not be implemented in the languages that libraries it uses are implemented in. For example, suppose there is a scientific package A written in Fortran, an algorithms package B written in Ada, and a mathematical library package C written in Fortran. If A depends on B and C, but B also depends on C, then both A and B will both have dependencies on Fortran standard libraries that are transitively implied, even though package B contains no Fortran, and package A does.

Dependency metadata is therefore insufficient for determining language use.

One strategy for overcoming this problem is to use metadata at a finer level of detail, namely source code filenames. In the remainder of this chapter, the procedure for producing

[1] http://old-releases.ubuntu.com/ubuntu/pool/main/o/openssl/openssl_0.9.8g.orig.tar.gz

a random selection of multilanguage packages is detailed. The procedure is to extract all filenames from all Ubuntu source packages, build a filename classifier to identify programming languages, use the classifier to eliminate non-multilanguage packages, and then manually verify whether or not randomly-selected candidate packages are actually multilanguage.

An initial attempt was made to use the Unix `file(1)`[1] tool to classify files by their contents. However, estimates from timing trial runs found that the tool would take approximately 2 days to classify all source files in Ubuntu. This estimate was based on using the tool via an in-process shared library instead of as an external tool invocation. Even then, the classifications produced would not be of much value for programming language identification. The `file(1)` tool only looks at the first 4KiB of a file, and many Java and C/C++ source files begin with more than 4KiB of license headers and copyright notices in comments. Since Java and C/C++ support the same comment characters, such files are indistinguishable to `file(1)`. Since this tool for inspection of file contents seemed ill-suited for the purposes of this study, a custom tool to classify files based on their names was developed instead.

3.1 Filename classification

The first step in classifying filenames is to download the source code for every package in Ubuntu. The source code URLs are specified in various package metadata files: each Ubuntu release contains a top-level file called `Release`[2] containing the names of all the different `Sources`[3] and `Packages`[4] metadata files describing source packages and binary packages, respectively. The metadata `Directory` and `Files` entries can be combined to form a URL to access the source code directly from a Ubuntu distribution site. Additional field columns contain file sizes and checksums for validation. In addition to original vendor-supplied source code, Ubuntu supplies metadata and Ubuntu-specific patches. In the case of `openssl` in Figure 3.1, the three files listed are the original 3.4 MB `openssl` source code archive;[5] a 1.4 KB signed copy of the metadata file;[6] and a 62 KB compressed patch file containing the combined Ubuntu patches and build scripts.[7]

It is usual practice for Ubuntu to distribute unmodified source code packages from the software vendor separately from Ubuntu-specific separate patch files. In 266/16 153 ≅ 1.6% of packages, the source code package is already modified by Ubuntu. Patch files are not used in the selection process. Ubuntu is used in the selection process for convenience—

[1]ftp://ftp.astron.com/pub/file/
[2]http://old-releases.ubuntu.com/ubuntu/dists/karmic/Release
[3]e.g., http://old-releases.ubuntu.com/ubuntu/dists/karmic/main/source/Sources
[4]e.g., http://old-releases.ubuntu.com/ubuntu/dists/karmic/main/binary-amd64/Packages
[5]http://old-releases.ubuntu.com/ubuntu/pool/main/o/openssl/openssl_0.9.8g.orig.tar.gz
[6]http://old-releases.ubuntu.com/ubuntu/pool/main/o/openssl/openssl_0.9.8g-16ubuntu3.dsc
[7]http://old-releases.ubuntu.com/ubuntu/pool/main/o/openssl/openssl_0.9.8g-16ubuntu3.diff.gz

`.tgz`	`.tar.gz`	`.tar.bz2`	`.tar`	`.lzma`	`.zip`	`.jar`	`.war`

Table 3.2: Compressed archive file extensions

detailed easily-parsable metadata, a large number of packages, and a collection of package versions that have been tested to be compatible with one another. However, the goal of this thesis includes understanding build systems for multilanguage software packages from the perspective of the developers of those software packages, which requires looking at the source code released by those developers rather than as later modified by Ubuntu. Therefore, Ubuntu-provided patches are not applied.

A tool called `mosd`, short for Mining Open-Source Distributions, was created as part of this thesis for extracting and analyzing metadata and filenames from Ubuntu's source code packages. It is a Java program that links via JNI to the open-source `libarchive` archive and compression library written in C. Note that the `lzma` tool must be installed for `libarchive` to handle `.lzma` files. Each downloaded Ubuntu source archive in uncompressed and the names of the files it contains are extracted. Compressed archives containing compressed archives with filenames matching entries in Table 3.2 are processed recursively. If archives cannot be decompressed by `libarchive` then they are treated as regular files. Such invalid compressed files occur, for example, as test inputs included with the source code for decompression programs.

There are 6 267 094 source files totalling 101GB, with 2 724 429 unique filenames. On a Late 2009 13" MacBook Pro with full-disk encryption, it takes about 30 minutes to process all the archives and extract all of the filenames. The filenames are encoded into a 417MB file; however, since filenames are largely highly-compressible text, the list of all filenames compresses with `gzip` compression to a 64MB cache file, which takes only 3 seconds to decompress. The compressed cache file takes under 20 seconds to load into memory, and then under 3 seconds for analysis code in `mosd` to iterate over all filenames in all packages. `mosd` supports reloading changed versions of its own Java classes on the fly, eliminating the need to wait for the cache to be reloaded after most code changes. One current limitation of `mosd` is that the in-memory representation of filenames is much less efficient than the on-disk structure, requiring multiple gigabytes of RAM.

After the filenames are extracted from the source code, each filename is classified based on its extension as being one of three types of files: source code in a specific programming language listed in Table 3.3; a known non-programming-language file type listed in Table 3.4; or an unknown type of file such as the ones in Table 3.5. Note that C and C++ are counted as the same programming language 'C/C++' for multilanguage package selection purposes. `Makefiles` and other build system scripts are counted as non-source-code since the goal is to identify implementation languages. Also, as a special case, `.in`, files, which

Language	Extensions
Ada	.ada, .adb, .ads
Asm	.asm, .s, .S
AWK	.awk
.bat	.bat
Boo	.boo
C/C++	.C, .C, .cc, .cpp, .cxx, .c++, .h, .H, .hh, .hpp, .hxx, .hin, .i
C#	.cs
ELisp	.el
Erlang	.erl
Fortran	.f, .F, .f90, .F90
Groovy	.groovy
Haskell	.hs
Java	.java
JS	.js
Lex	.l, .ll
Lisp	.lisp, .lsp
Lua	.lua
M	.m
MetaFont	.mf
ML	.ml, .sml, .mli
Pascal	.pas
Perl	.pm, .pl, .PL
PHP	.php, .phpt
Pike	.pike
Python	.py
R	.R
Ruby	.rb
Scala	.scala
Scheme	.scm
sci	.sci
sed	.sed
SQL	.sql
TCL	.tcl
TeX	.sty, .dtx
Visual basic	.vb
XSL	.xsl
YACC	.y

Table 3.3: Filename classification: Programming languages

Build scripts	MANIFEST	**Image files**	**Version control**
makefile	NEWS	.gif	Entries
Makefile	PORTING	.jpg	Repository
GNUmakefile	PORTS	.jpeg	Root
Imakefile	README	.png	Tag
rules	THANKS	.bmp	.cvsignore
.ac	TODO	.eps	.gitignore
.am	VERSION	.svg	.svn-work
.guess	changelog	.svgz	.svn-base
.m4	Changelog	.xpm	.kndx
.mk	ChangeLog	.xbm	.knit
.mak	CHANGELOG	.tga	**Other**
.make	**Translations**	.dot	.odt
.spec	.po	.psd	.ppt
.sh	.gmo	.pdf	.mpg
.cmake	.pot	.tiff	.csv
.jam	.mo	.tif	.bsp
.sub	**Documentation**	.ico	.swf
.pro	.tex	.xcf	.dxf
Generated files	.info	**Sounds**	.vcprog
configure	.texi	.mp3	control
depcomp	.texinfo	.wav	Change
install-sh	.pod	.gsm	.strings
missing	.man	.ogg	8gb-1
Text	.1	**Object code**	big
.txt	.2	.so	.md5
.TXT	.3	.elc	.vcproj
.rst	.4	.pyc	.ui
ABOUT-NLS	.5	.dll	.desktop
AUTHORS	.6	.pdb	.cache
BUGS	.7	.class	.conf
CHANGES	.8	**Fonts**	.properties
COPYING	.9	.ttf	.yaml
COPYRIGHT	.Rd	.otf	.sdf
copyright	.docbook	.bdf	.txt,v
FAQ	.htm	.pfb	.ppd
FILES	.html	.vf	.ics
INSTALL	.sgml	.afm	.rgb
LICENSE	.css	.tfm	.pk3

Table 3.4: Filename classification: Non-source code

Extension	File count	% of all files
`.xml`	127 923	2.04%
`.t`	24 730	0.39%
`.d`	22 137	0.35%
`.dtd`	17 350	0.28%
`.idl`	14 755	0.24%
`.def`	14 244	0.23%
`.map`	13 074	0.21%
`.dat`	12 989	0.21%
`.inc`	12 883	0.21%
`.out`	11 788	0.19%
⋮	⋮	⋮

Extension	Byte count	% of all files
`.xml`	1 472 700 398	1.49%
`.dfsg`	503 952 752	0.50%
`.ps`	493 771 444	0.49%
`.dat`	435 060 818	0.43%
`.ts`	374 998 587	0.37%
`.map`	337 066 123	0.33%
`.bin`	237 519 131	0.24%
`.0`	228 689 365	0.23%
`.out`	198 667 107	0.20%
`.dic`	190 449 437	0.19%
⋮	⋮	⋮

Table 3.5: Top unknown filename extensions, by file count and byte count

```
packages ← list of all Ubuntu source packages in alphabetical order

{ create a sample of packages weighted by language count }
sample ← new List
for package in packages do
  languageCount ← programming language count for package, using Table 3.3
  weight ← languageCount ≥ 2 ? languageCount : 0
  for i = 0; i < weight; i++ do
    sample.add(package)
  end for
end for

{ build up a permuted list of packages }
permutation ← new List
random ← new java.util.Random(seed = "multi language systems".hashCode())
while permutation.size() < packages.size() do
  package ← sample[random.nextInt(sample.size())]
  if package ∉ permutation then
    permutation.add(package)
  end if
end while

{ select 5 case studies }
numCaseStudies ← 0
i ← 0
while numCaseStudies < 5 do
  candidate ← permutation[i]
  if manualInvestigationShowsPackageIsMultilanguage(candidate) then
    selectForCaseStudy(candidate)
    numCaseStudies ← numCaseStudies + 1
  end if
  i ← i + 1
end while
```

Figure 3.3: Candidate selection algorithm

are usually template files, and `.gz` and `.bz2` files, which are usually compressed versions of other files, have their suffixes stripped before classification. For example, `Makefile.in` and `background.bmp.gz` are classified correctly as a Makefile and a bitmap image file, respectively.

The classification tables were developed by manually adding rules for filename patterns until 80% of all files, measured both by file count and byte count, had a known classification. The remaining 20% of files are largely data files and project-specific files that would require more detailed analyses to classify. The top unclassified filename patterns are shown in Table 3.5. A target of 80% classification was chosen as an acceptable tradeoff between classifier accuracy and research needed to improve accuracy. It is high enough to help identify a variety of useful case studies, without requiring too much research on an issue not directly relevant to build systems. This is discussed in more detail in Section 3.4

3.2 Multilanguage status verification

Manual examination of candidate packages is necessary because packages containing source files in different languages are not necessarily multilanguage. Filename classification can eliminate many non-multilanguage packages, but cannot determine how the languages are used or whether the packages are multilanguage in the sense of using multiple interdependent implementation languages. There are 10 903 packages that cannot be multilanguage because, according to filename classification, they contain fewer than two programming languages. For the remaining 5 250 candidate packages, the following procedure is used. A previously unselected package is selected at random, but with each package's chance of being selected weighted by identified language count. The intent of weighting is to increase the likelihood of selecting a multilanguage package. Each selected package is manually investigated to determine whether it is multilanguage. This continues until a sufficient number have been found for further study. The details of this selection procedure are given in Figure 3.3, which includes the `java.lang.Random` seed-value for reproducibility. The identified language counts for the selected packages are shown in Table 3.7. The packages examined, the languages found, and the multilanguage status determination is shown in Table 3.6.

Why do packages turn out to not be multilanguage even though filename classification finds many different programming languages? The specific reasons for specific packages are listed in Table 3.6, but in general one or more of the following reasons applies:

- Only one implementation language. There are source files in several languages, but only one language is used in the implementation. Other languages are used for purposes such as developer scripts, bootstrapping, sample code, testing, and so on.

  One prevalent reason for a package with source files in multiple languages to have

Package	Implementation languages	Multilanguage?
`db4.6`	C	No. Written in C, with Java bindings, and with Tcl scripts for unit tests.
`yacas`	C, Java	No. Completely separate implementations in two different languages.
`synopsis`	C, Python	Yes. Python user interface with some C/C++ components.
`python3.0`	C, Python	Yes. Python interpreter written in C, many standard libraries written in Python.
`libdb-sqlite2-perl`	C	No. Some binding subroutines written in Perl.
`gears`	C	No. Written in C/C++, with sample code in other programming languages.
`zope-external-editor`	Python	No. Contains one DOS batch file.
`gnat-gps`	Ada, C, Python	Yes. Mainly Ada, with C/C++ utility functions, and part of UI written in Python.
`axiom`	C, Lisp, Scratchpad	Yes. Scratchpad compiler written in C and Lisp with math library written in Scratchpad.
`supertux`	C	No. Game data is stored in Scheme-like data files, but there is no Scheme code.
`libunwind`	C	No.
`penguin-command`	C	No. 11-line Objective-C wrapper for Mac OS version.
`fpc`	Pascal	No. All Pascal. Some developer utilities and tests in C.
`eb`	C	No. Some developer Perl scripts.
`rtai`	C	No. Has bindings for Scilab.
`libparse-recdescent-perl`	Perl	No. Parser comes with sample C file to demonstrate parsing.
`plplot`	C	No. Bindings for 9 other languages.
`ruby-prof`	C, Ruby	Yes. Mainly Ruby with profile data collection module in C.

Table 3.6: Multilanguage status of candidate packages

▨ Highlighted rows correspond to packages identified as multilanguage.

	db4.6	yacas	synopsis	python3.0	libdbd-sqlite2-perl	gears	zope-externaleditor	gnat-gps	axiom	supertux	libunwind	penguin-command	fpc	eb	rtai	libparse-recdescent-perl	plplot	ruby-prof
Ada								1180										
Asm	1		14	38		1			4634		31		2		1			
AWK	13					6		1	15									
.bat	1			31		1	1		27				6				2	
C/C++	611	131	527	534	44	1725		338	7428	478	415	31	64	99	428	1	264	10
C#	41																	
ELisp				1					46									
Fortran								1	3								40	
Java	327	94				10		1									34	
JS		6	13			109		1	1								1	
Lex			1					8	351	1			8		1			
Lisp									1909									
M				6		5			6			1			3		177	
ML																	32	
Pascal													1717					
Perl	15	3			4			3	57	2			3	1		66	42	
PHP	2					28									1		8	
Python			168	1545		94	18	78		1							61	
R	1								37									
Ruby																		35
Scheme									9	1								
sci															43			
sed	1							1	62				6				2	
SQL						2							3					
TCL	378					23		13	103				1				62	
TeX									19				35					
XSL			12										1		3		2	
YACC						1		3	24				2		1			

Table 3.7: File counts per automatically-identified language for candidate packages

only one implementation language is through the presence of bindings. *Bindings* are interfaces that allow software written in one language to be used by software written in another language. For example, a user-interface widget package implemented in C may provide an interface for Python scripts to use the widgets as well. Such bindings are often automatically generated by tools like the Simplified Wrapper and Interface Generator, also known as SWIG [7].

Bindings within a single package, even automatically-generated ones, may provide the link between parts written in different languages that make the package multilanguage.

Package	Description	Languages	Industrial?	Age (years)	KSLOC	Committers
`synopsis`	Source code documentation	C++ Python	No	9	94	5
`python3.0`	Programming language	C Python	No	18.5	585	150
`gnat-gps`	IDE	Ada Python C	Yes	5	450	17†
`axiom`	Computer algebra system	Lisp Scratchpad C	Yes and No	30+	360	41†
`ruby-prof`	Profiler	Ruby C	No	2.5	4	4

† Undercounts—`gnat-gps` and `axiom` have only 1 and 5 years, respectively, of public commit history

Table 3.8: Selected packages

However, when a package written in one language provides bindings to allow use from a second language, without itself using the second language for anything other than implementing the bindings, the package is not considered multilanguage. The distinction is whether the package uses multiple languages in its implementation. For example, a user-interface widget package written in C that provides Python bindings but does not use them is not multilanguage. But if the package implements some of its widget functionality in Python code via the bindings, then it would be multilanguage.

- No integration. For example, the `yacas` package contains a computer algebra system implemented in both C++ and Java, but there are actually two independent parallel implementations: Yacas and JavaYacas.
- Noise. An over-zealous attempt to classify all possible programming languages resulted in a filename classifier that identifies many files such as DOS batch files, sed scripts, and TeX files that, while technically files written in Turing-complete programming languages, use languages that lack intra-process communication mechanisms necessary to participate in multilanguage relationships. These classifications increase the language count without any relevance towards whether the package is multilanguage.

The resulting selection of packages is shown in Table 3.8.

3.3 Discussion

The goal of this chapter is to systematically select multilanguage software packages in order to perform case studies of their build systems. This is achieved. There are two specific claims made:

1. The five packages in Table 3.8 are reasonable subjects for case studies.

2. Using filename classification followed by manual inspection is an appropriate way to systematically select case study subjects with desired properties.

Are these claims adequately justified? Are there faster, easier, or otherwise better ways to get these results?

First, evaluation of case study subject selection is determined from the results of the case study itself, not of the selection procedure. Qualitative studies, including this thesis, use analytic inference and generalization, not the statistical inference and generalization of experimentation [73]. In fact, it is recommended [52] that case studies not use random sampling, but instead purposive sampling: the cases selected are specifically those considered most likely to illuminate a subject or to confirm or refute theories. The driving concern in selecting case studies is not that cases may not be strictly representative, but that they may be uninteresting or so similar that no useful comparative conclusions may be drawn from analyzing them.

However, this is largely an exploratory work—there is no previous research that focuses specifically on build systems for multilanguage software. Not enough is known about build systems for multilanguage software to be able to perform purposive sampling. For this study, therefore, random selection is used to produce an initial set of cases for exploratory analysis. The selection is diverse along many dimensions, as detailed in Table 3.8:

Variety of application domains, including automated documentation processing, programming language implementation, computer algebra, and an interactive development environment.

Various programming languages in different combinations, including C/C++, Ada, Python, Ruby, Lisp, and Scratchpad.

Mix of industrial and non-industrial packages. `gnat-gps` is a commercial product of AdaCore, Inc. `axiom` was a commercial product of IBM and then the Numerical Algorithms Group, and was open-sourced at its commercial end-of-life. Most of the source code was industrially developed, but the current build system was written by open-source volunteers.

Variety of ages of packages, measured in years from the first release of the package to the release of the version included in Ubuntu 9.10 and studied here.

Various sizes from the tiny `ruby-prof` to the 500+ KSLOC `python3.0`. Size is measured in Kilo Source Lines of Code, i.e., number of 1 000s of non-blank non-comment source code lines.

Various team sizes. As a proxy for team size, the given measurement is the number of distinct usernames that committed to the source code repository up to the studied release. Note that the person committing changes is not necessarily the author of the changes. Complete commit history is not available for `axiom` and `gnat-gps`.

Also, these figures reflect some manual dealiasing of the usernames for 14 different `python3.0` committers who have committed source code under multiple usernames, such as the usernames `guido`, `gvanrossum`, and `guido.van.rossum` all belonging to Python creator Guido van Rossum. There may be other such collisions, but these 14 are straightforward to identify because they involve similar usernames with high numbers of commits. For example, without dealiasing, the list of usernames sorted by commit counts contains `guido` at #1 and `gvanrossum` at #4. Manual inspection of usernames indicates that the other projects do not appear to have multiple usernames for individual committers in the data used for this table.

However, there is one dimension along which the packages lack diversity: while each package addresses a different application area, they are all tools for programming languages—source code documentation, an IDE, a profiler, a general-purpose programming language, and a computer algebra system that implements its own programming language. This may be just a coincidence; it may be that most packages in the open-source ecosystem in general or Ubuntu in particular are programming tools; it may be that open-source multilanguage software packages tend to be programming tools; or it may be bias in the selection process, the potential for which is discussed in the next section. In any case the criteria for whether this sample is appropriate or not is whether the packages are different enough to draw useful conclusions from comparison in future chapters.

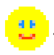
The second claim is that filename classification followed by manual inspection is appropriate for identifying multilanguage packages. Filename classification removes more than 10 000 packages from consideration, but leaves 5 250 other packages requiring manual inspection. This manual effort can be managed with random selection, but might it be possible to automate verification as well? The procedure used in this chapter counts either a single file or a thousand files equally toward the total language count in a package. Could some sort of threshold be used? It appears not. Consider the `gnat-gps` package. At shown in Table 3.7, there are more than a thousand Ada files and only 78 Python files—less than 5% of the source code. From this number, it seems that Python may play an insignificant role in the package. However, detailed investigation in the next chapter shows that Python is a critical language in this software, and is the preferred language for adding new GUI features. With such a small amount of source code playing a critical role in this package, it does not seem that automated thresholding on file counts or sizes alone can properly distinguish multilanguage packages. The presence of multiple languages is necessary but not sufficient for a package to be multilanguage.

3.4 Threats to validity

This section addresses factors that may affect the validity of the claims discussed in the previous section. The factors are organized into four components: construct validity, internal validity, external validity, and reliability [73].

Construct validity—are the various measures used measuring what they are intended to? The primary question here is whether filename classification is measuring language use correctly. The filename patterns used cover 80% of the source files in Ubuntu, measured both by file count and byte count. This is an arbitrary threshold. What is in the other 20%? The top remaining unknown file types are shown in Table 3.5. The most frequent file type is XML, which is not so much as an unknown file type as it is a known file type that cannot be definitively classified as 'source code' or 'not source code' based on filename alone. It is a generic file format usually used for pure data but which can contain literally anything, including source code in XML syntax such as the build scripts for Ant's `build.xml` programming language. Similar issues apply to other top unknown file types, but also to more common languages: some C files are actually XPM-format image files as in Figure 3.4; and JavaScript is often used as a syntax for storing data in JavaScript Object Notation. Lisp takes the idea even further, mixing data and code throughout source code but also treating code as data when convenient. The filename classifier treats C and JavaScript as programming languages because that is what they were designed for, and leaves XML files

```
/* XPM */
static char *smile[] = {
    "18 18 6 2", /* columns rows colors chars-per-pixel */
    "   c None",          ",, c yellow",        "## c orange red",
    "@@ c DodgerBlue1",   ".. c honeydew3",     "-- c light salmon",
    /* pixels */
    "                                    ",
    "            ,,,,,,,,,,,,            ",
    "        ,,,,,,,,,,,,,,,,,,,,        ",
    "      ,,,,,,,,,,,,,,,,,,,,,,,,      ",
    "    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,    ",
    "    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,    ",
    "  ,,,,,,,,,,..@@,,,,@@..,,,,,,,,,,  ",
    "  ,,,,,,,,,,@@@@,,,,@@@@,,,,,,,,,,  ",
    "  ,,,,,,,,,,..@@,,,,@@..,,,,,,,,,,  ",
    "  ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,  ",
    "  ,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,  ",
    "  ,,,,,,,,,,##--,,,,--##,,,,,,,,,,  ",
    "    ,,,,,,,,--########--,,,,,,,,    ",
    "    ,,,,,,,,,,,,,,,,,,,,,,,,,,,,    ",
    "      ,,,,,,,,,,,,,,,,,,,,,,,,      ",
    "        ,,,,,,,,,,,,,,,,,,,,        ",
    "            ,,,,,,,,,,,,            ",
    "                                    "
};
```

Figure 3.4: Pixmap example for . The textual file format, which is supported by image editors and viewers, is also valid C source code that can be compiled into a program so that icons are available in memory immediately on program startup [32].

as unknown because XML was not designed to be a programming language though it is commonly used as one.

There is more that could be done to distinguish the uses of different file types, but it would be a lot of work that would not reveal anything about build systems. Possibilities requiring examination of file contents instead of just names include comparing root elements of XML documents, looking at `#!`-interpreter lines in files without extensions, and counting function definitions in JavaScript files to distinguish code from data. However, this would involve either creating or acquiring and integrating fact extractors and dynamic analysis tools for many different languages. Even then, many single-purpose language uses would be missed. For example, the `axiom` package contains 1 349 `.pamphlet` files of literate source code of C, Lisp, and Scratchpad, all embedded in LaTeX, but there are no other `.pamphlet` files in any other Ubuntu package. While it could be possible to develop tools to detect this, such as by scanning through source files looking for C-like or Lisp-like strings—it would be unlikely to provide additional insight about build systems.

As for the 80% threshold, a similar process of classifying files based on filename performed by Robles [54] had classification rules resulting in 6% of *commits* containing unknown file types. This is not necessarily significantly different from 20% of *files* being of unknown type, because it is likely that changes to the more-numerous files with common extensions, such as `.c` and `.java`, occur more frequently than changes to less-numerous files of rarer file types. Another related study by Karus and Gall using filename classification for programming language identification [36] does not quantify the proportion of classified and unclassified files.

However, there is an issue with the random selection procedure. In an attempt to reduce the number of packages requiring manual inspection, the packages are weighted by identified language count when performing random sampling, on the assumption that multilanguage packages, in general, use more programming languages than non-multilanguage packages. This weighting or biasing increases the likelihood that the manually inspected packages are multilanguage, but it also biases in other unknown ways. One possible bias is that, due to the file classification's inclusion of languages such as sed and Awk that cannot participate in multilanguage relationships because they lack intraprocess communication mechanisms, the selection is biased towards old-fashioned Unix programs that tend to have sed and Awk scripts in their source distributions. This may be why, for example, there are no packages using Java in the selection. This issue was not identified until after the detailed case studies had been performed.

Internal validity—are discovered relationships more than just coincidence? Were the packages identified as multilanguage actually multilanguage? Manual investigation and detailed case studies to come show that this is not the case. However, it is possible that multilanguage packages using languages not identified by the filename classifier were incorrectly pruned from consideration as candidates. As the goal is to find an appropriate selection of multilanguage packages, not to identify every multilanguage package, this is not a serious threat to validity of the study as a whole.

External validity—how generalizable are the findings? The specific selection of case studies is not intended to be generalizable to other studies. The details of the selection process will hopefully be useful to other researchers for other studies. Filename analysis is helpful for identifying candidate multilanguage packages. However purposive selection for coverage along important dimensions should be used rather than uniform random sampling or sampling intended to be random but actually biased in unknown ways.

Reliability—could someone else do this and get the same result? The full details of the selection process are presented here in sufficient detail to reproduce it, including tables of filename classifications and initial random seed values. There is some possibility for getting different results: a different version of `libarchive` may successfully decompress a different set of malformed archive files, resulting in different language counts and changing the sample; there could be disagreement over whether a candidate package is truly multilanguage or not, but the multilanguage status of the packages examined in Table 3.6 seems unlikely to arouse controversy.

Conclusion The selection of case studies and the process that produced them are both valid for an exploratory case study. Random selection must be used with care in qualitative studies.

3.5 Related work

Other researchers use Ubuntu metadata, filename classification, or both, in related studies.

In his doctoral dissertation, Robles [54] shows that exhaustive analysis of thousands of software projects is possible, and that socio-technical analysis is required in addition to purely technical analysis. One aspect of that is an extended discussion of data sources and formats for libre software research. The use of Ubuntu metadata and package files in this thesis is based on that discussion.

There are two related studies in Robles's dissertation. The first uses filename extensions to classify source files into one of seven categories: developer source code, developer documentation, user documentation, version control overhead, translations, user interface files, and sound/video files. The categories are used to identify communities of open-source contributors with different roles such as translators, technical writers, or designers. Robles lists the filename extensions used in an appendix. In contrast, this thesis examines only the source code category but further classifies the files in that category by programming language.

The second related study in his dissertation uses a third-party tool to gather statistics about the number, size, and programming language of source code for all of Ubuntu. The language statistics are aggregated across the entire operating system distribution and changes between released versions are then analyzed. In contrast, the selection process in this chapter examines only a single version of Ubuntu, but with per-project language statistics rather than statistics aggregated across the entire distribution.

Robles also co-authored a later article [28] that demonstrates the utility of package metadata for a variety of purposes, such as identifying competing packages or evaluating packages by checking whether they depend on known buggy, insecure, or unmaintained packages. German [27] uses similar techniques to define "dependency success": packages like `iconv` and `curses` can be considered highly successful because of how many other packages depend on them. These studies use package metadata to infer package statistics that are invariant under transitive reduction of the dependency graph. Examining only metadata does not suffice for the selection process in this chapter because language use is not such a statistic.

Karus and Gall [36] analyze trends of languages that are used together in order to better understand the role of XML. They find, for example, that nearly all Java developers use XML, but only half of C developers. Their method examines each historical revision of a collection of open-source projects. All files committed together [22] are classified by programming language or file type. This produces statistics about how often developers work with specific different languages or file types at the same time. The selection process in this chapter examines the files in released software versions rather than each intermediate revision, but for all source files in Ubuntu rather than for a collection of 22 projects. Both Karus and Gall and this chapter classify programming languages based on filenames, but, due to the much larger number of files to be examined and peripherality of XML for the selection process, this chapter does not examine file contents the way that Karus and Gall do to subclassify different types of XML files.

3.6 Conclusion

This chapter gives an overview of the metadata associated with Ubuntu packages, shows that metadata is insufficient for language identification, describes a combination of automated and manual procedures for selecting multilanguage packages, and produces a selection of cases to study in the next chapter.

Additionally, the procedure used and the tools developed can be adapted for purposive selection in other software case studies. Filename patterns can be used to identify software packages using specific programming languages, tools, or file formats. While it is unnecessary and tangential to do so when selecting multilanguage packages for case studies of build systems, the `mosd` tool could also be extended to examine not only filenames but also the contents of specific file types. File contents could be used to extract software package properties for purposive selection in other case studies.

Chapter 4

Build system case studies

The procedure used for the case studies is for the investigator to build each package using the vendor-supplied build system, while observing any issues encountered and examining additional data sources such as source code, documentation, and mailing list archives in order to determine the following about each package and its build system:

- Purpose and abstractions: What does the package do? What are the major abstractions?
- Architecture, languages, and interactions: What are the major parts of the software, what languages are they written in, and how do they interact?
- Build system: What is the structure of the build system and how is it implemented?
- Build issues: What, if any, issues are there in building the package?
- Rebuild issues: What, if any, issues are there in building changes to the package?

The first three questions address the perspective of end-users, developers, and build system developers, respectively, while the last two questions on build system issues are especially relevant to developers but also relevant to end-users who may build the software from source code.

An additional question:

- Build features: What build system features prevent build problems similar to those encountered in other packages?

describes issues *not* experienced, compared to other case studies, and how they are avoided. Some of these observations may seem incomplete, especially when they refer to case studies that come after. However, the issues flagged with numbered icons ⚠1 , ⚠2 , ⚠3 , and so on, as well as features flagged with numbered icons ①, ②, ③, and so on, are referred to in the following chapter when the findings from the case studies in this chapter are compared and synthesized into build patterns and anti-patterns.

There are a few more details to cover about how the case studies are conducted.

The vendor-supplied build system is used, rather than the Ubuntu-specific patches and build scripts that can automatically produce binary packages. This is done to support several common use cases such as: an end-user installing a package they have downloaded the source code for; a developer beginning to make their first changes to a package they use regularly and would now like to modify; and a build expert examining a new package with an eye towards improving its build system. Developers, including build system experts, will usually want their changes integrated into future official releases of the software, and submitting changes for inclusion usually requires patches against official releases, not against Ubuntu-modified versions. However, end-users will typically install Ubuntu-supplied precompiled binaries rather than using any build system.

Ubuntu's package-management tools are used to automatically install each package's build-and run-time dependencies prior to build. This permits the study to focus on package-specific build system issues, rather than issues such as missing dependent libraries and libraries, or incompatible tool, library and package versions. These are issues which Smith [60, Ch. 15] identifies as some of the top issues for practitioners in building open-source software, but which build system designers have little or no control over. For those encountering this issue, the solution is straightforward though tedious and potentially frustrating: install compatible versions of all required libraries, tools, and prerequisite packages.

When first building each case study package, the expectation is that the package contains a single build script and optionally an automatic configuration script in the root directory of the source code, and that the build system "just works"—executing the configuration and build scripts quickly results in an executable binary without requiring any manual intervention or source code modification. In general, in order to reduce the delay in the feedback cycle from changing code to seeing the results in a running program, there is also an expectation that the software can be run and tested directly from the build directory, without requiring installation. Furthermore, once built, changing any source files and re-executing the build system should quickly result in an updated executable binary. The following case studies examine how this expectation matches with reality.

Diagrams

To help explain the answers to the case study questions, two types of architecture diagrams are also given for each system: multilanguage architecture diagrams and build-time view diagrams. The common notation for these diagrams is shown in Figure 4.1.

Multilanguage architecture diagrams are logical architectural view diagrams [39] that

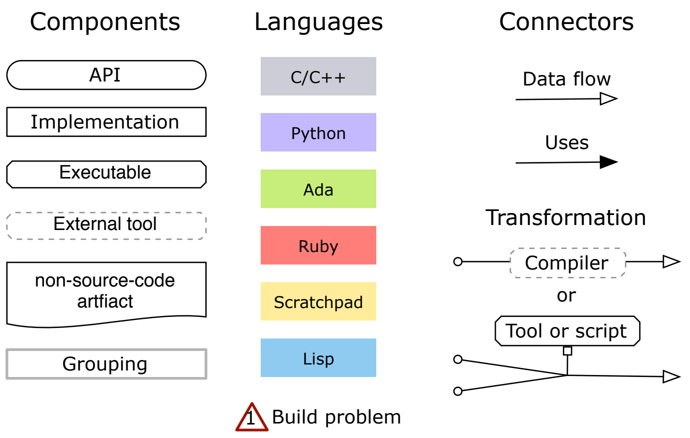


Figure 4.1: Notation legend for architecture diagrams, used in Figures 4.3, 4.4, 4.5, 4.7, 4.10, 4.11, 4.12, 4.16, 4.20, 4.24, and 4.25.

display the major parts of the software that are written in different programming languages, and show how the parts written in different languages interact.

Build-time view diagrams are a view of the software architecture as it appears at build time. The diagrams in this thesis use a modified version of the notation defined by Tu and Godfrey, who introduced the concept of build-time view diagrams [67, 68]. The build diagrams in this thesis focus on the structure of the build rather than implementation details of the build system. That is, source artifacts, build products, and the compilation tools used to derive build products are shown, rather than the build scripts which implement the build process. Additionally, many implementation details such as which particular C compiler is used are abstracted away. Such diagrams are divided into three stages vertically: the source view, the build view, and the execution view. The source view displays source artifacts shipped with the original source code. The build view shows intermediate artifacts and the build tools that transform artifacts. The execution view shows how build products are linked together at runtime. The left-to-right ordering of entities in these diagrams is not significant. The transformations could be performed in any order, or in parallel, although many build transformations will fail or produce inconsistent build products if transformations are run before their inputs exist, or if inputs change after transformations are executed.

The build-time view diagrams are annotated with symbols showing where the encountered build problems occur. These symbols usually mark where the symptoms of the build problem are noticed, rather than any root causes for the problems.

4.1 Case study: `synopsis`

A summary of the findings for `synopsis` is given in Table 4.1. The multilanguage diagram is Figure 4.3 and the build-time view diagram is Figure 4.4.

Question	Findings
Purpose and abstractions	Source code documentation tool using pipes and filters
Architecture, languages, and interactions	Pipes and filters architecture. Mostly Python, but with parser modules written in C/C++ that create Python data structures. These are modified copies of third-party parsers
Build system	`distutils` with independent synopsis-created autotools build systems for documentation and C/C++ parser modules
Build issues	⚠1 Running built software requires manual path setup ⚠2 Documentation build fails on case-insensitive filesystems ⚠3 PDF documentation does not build due to problem in third-party tool
Rebuild issues	No issues
Build features	① Uses automatic dependencies ② Uses shared libraries ③ `distutils`-based build with autotools subbuilds ④ Build aborts immediately on error
Conclusion	Only major issues: path configuration, scalability

Table 4.1: Summary of findings for `synopsis`

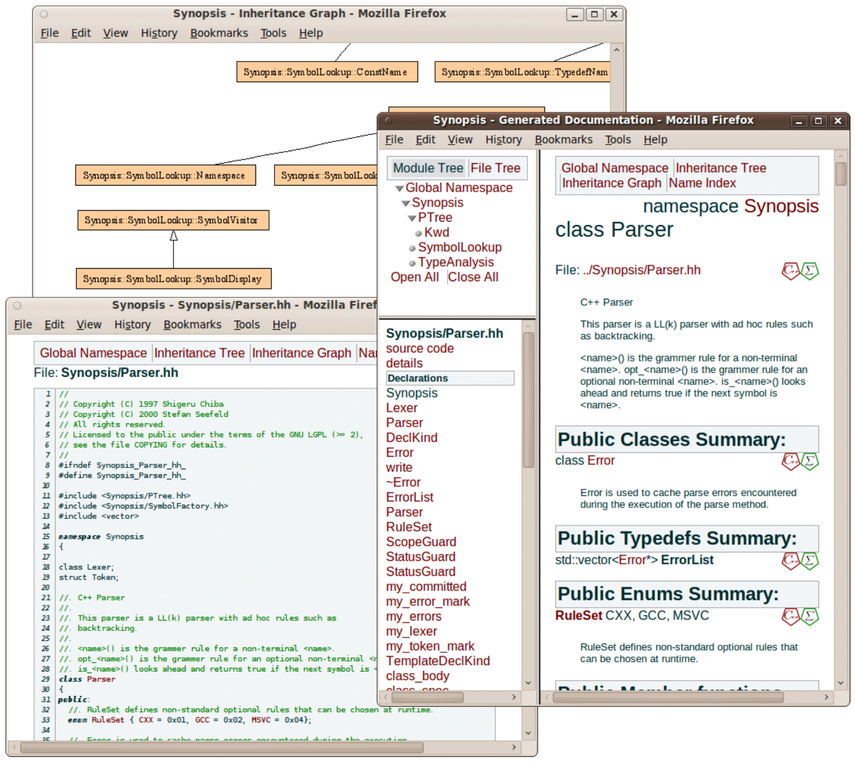
What does the package do? What are the major abstractions?

`synopsis` is a source code documentation tool. It extracts symbol declarations and specially-formatted comments from source code in order to produce documentation and class diagrams as shown in Figure 4.2. It has extensible language support, currently supporting Python, C, C++, and IDL source code. It also supports several different output formats including HTML, DocBook, and Texinfo.

`synopsis` was created as part of an experimental CORBA-based UNIX windowing server called Berlin/Fresco [35]. Berlin/Fresco uses higher-level interfaces than typical display servers such as X11; while a typical X11 command from an application to the display might be, "draw a line," a typical Berlin/Fresco command is "create a window" or "create a button." In technical terms, Berlin/Fresco embeds the windowing toolkit in the display server. Berlin/Fresco is written in a combination of C/C++, Python, and IDL, and `synopsis` was created as a documentation tool to support all of those languages. Berlin/Fresco's website is now gone, and there have been no source code changes since 2005,[1] but `synopsis` is still maintained.

The abstraction presented by `synopsis` to the end-user is a network of parsers, transformers, and formatters. Instances of these components are connected together to form an end-to-end processing chain from textual source code to publishable documentation. Parsers extract symbol declarations and specially-formatted comments from source code. Transformers perform many different sorts of useful operations, such as parsing Javadoc

[1] http://web.archive.org/web/*/http://src.fresco.org/viewcvs/Fresco/



File: Synopsis/Parser.hh

```
//
// Copyright (C) 1997 Shigeru Chiba
// Copyright (C) 2000 Stefan Seefeld
// All rights reserved.
// Licensed to the public under the terms of the GNU LGPL (>= 2),
// see the file COPYING for details.
//
#ifndef Synopsis_Parser_hh_
#define Synopsis_Parser_hh_

#include <Synopsis/PTree.hh>
#include <Synopsis/SymbolFactory.hh>
#include <vector>

namespace Synopsis
{

class Lexer;
struct Token;

//. C++ Parser
//.
//. This parser is a LL(k) parser with ad hoc rules such as
//. backtracking.
//.
//. <name>() is the grammer rule for a non-terminal <name>.
//. opt_<name>() is the grammer rule for an optional non-terminal <
//. is_<name>() looks ahead and returns true if the next symbol is <
class Parser
{
public:
  //. RuleSet defines non-standard optional rules that can be chosen at runtime.
  enum RuleSet { CXX = 0x01, GCC = 0x02, MSVC = 0x04};
```

namespace Synopsis

class Parser

File: ../Synopsis/Parser.hh

C++ Parser

This parser is a LL(k) parser with ad hoc rules such as backtracking.

<name>() is the grammer rule for a non-terminal <name>. opt_<name>() is the grammer rule for an optional non-terminal <name>. is_<name>() looks ahead and returns true if the next symbol is <name>.

Public Classes Summary:

class Error

Error is used to cache parse errors encountered during the execution of the parse method.

Public Typedefs Summary:

std::vector<Error*> ErrorList

Public Enums Summary:

RuleSet CXX, GCC, MSVC

RuleSet defines non-standard optional rules that can be chosen at runtime.

Figure 4.2: `synopsis`-generated documentation

commands in comments, or access restriction, i.e., removing references to non-`public` declarations. Transformers pass their outputs to the inputs of other transformers until a final formatter is reached, which outputs documentation for viewing. Users can use standard combinations of parsers, transformers, and formatters from the command-line; users can create custom pipelines from the command-line with a sequence of `synopsis` calls that read from and write to intermediate files; they can also create custom pipelines using the Python API directly; and can also create their own parsers, transformers, and formatters in Python.

What are the major parts of the software, what languages are they written in, and how do they interact?

A diagram showing the architecture and implementation languages of Synopsis and its APIs is shown in Figure 4.3. The architecture closely mirrors the end-user abstractions described in the previous paragraph.

There is a Pipeline API for configuring the network of parsers, transformers, and formatters. This pipeline API, the user-interface, the Python parsers, and all of the processors and transformers are written in Python.

Each parser module take filenames and include-file search-path options as input, and returns annotated lists of symbol declarations. There is a parser module for each of C, C++, Python, IDL, and Cpp. "Cpp" is `synopsis`'s name for unpreprocessed C/C++

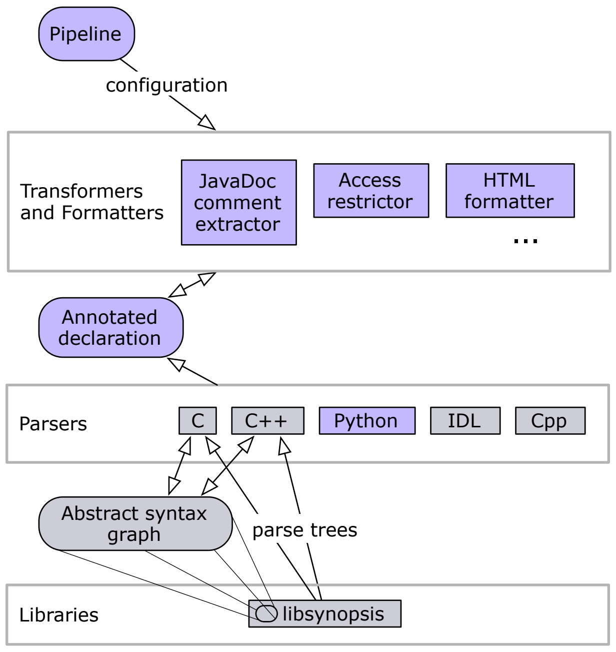


Figure 4.3: `synopsis` multilanguage diagram, using the notation of Figure 4.1. Parsers written in various languages extract declarations from source code. All subsequent processing is implemented in Python. Pipelines are configured, consisting of transformers that annotate declarations, and formatters that produce output for humans.

source code. A parser for this is needed to support macro documentation.

Each parser is derived from other open-source projects. The Python parser uses Python's builtin `compiler` and `tokenize` modules. The other parser modules all consist of Python interfaces to modified versions of third-party parsers written in C++: the C, C++, IDL, and Cpp parsers are based on ctool, OpenC++, omniORB, and ucpp, respectively. Each has a different mechanism for passing the extracted data from C++ to Python. The C++ parser builds an in-memory representation and then transforms it by calling the low-level CPython API directly. the C parser gets an in-memory representation from libsynopsis and then calls on a C++ transformation class which uses an object-oriented C++ wrapper `Python::` for the CPython API. The Cpp parser creates a Python data structure directly using `Python::`. Finally, the IDL parser uses an omniORB-authored Python wrapper for which the `synopsis` developers supply a Python class implementing the Visitor pattern [23].

The parsers for C and C++ source code use a common library called libsynopsis to generate full parse trees. Although only an annotated list of declarations is returned to external callers, the C and C++ parsers create a full Abstract Syntax Graph with limited support for rewriting source code. This is an artifact of an original design goal of `synopsis`, namely to some day become a general-purpose source code transformation tool.

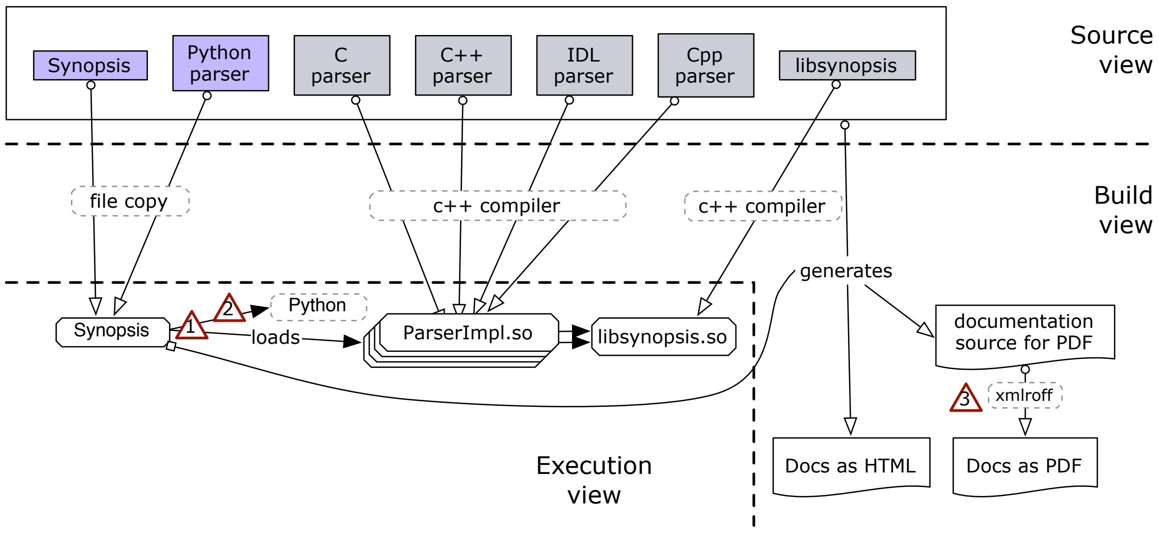


Figure 4.4: `synopsis` build diagram, using the notation of Figure 4.1. Executable Python source code is copied to the execution environment. Modules of C/C++ code are compiled to shared libraries, then loaded by `synopsis` at runtime. Documentation is produced by running `synopsis` on its own source code.

What is the structure of the build system and how is it implemented?

The overall build is driven by a Python `distutils` script, which is also responsible for packaging and installing the source. For building the C++ portions of the source, the `distutils` script in turn calls independent build systems for both the package's documentation and each C++ component: libsynopsis and each of the C, C++, Cpp, and IDL parsers. Each of these build systems produces a shared library. The shared libraries for the parsers are loaded by Python code. The shared library for libsynopsis is loaded by parsers that use it. The build system scripts are relatively simple, generally totalling fewer than 100 lines of autotools input. Each of these build systems is created by the Synopsis authors, rather than being created by the authors of the open-source projects from which each component is derived. One exception is the build for the Boehm garbage collector [9] used by and distributed with libsynopsis, which uses an autotools build system created by Red Hat.

What, if any, issues are there in building the package?

Three issues were encountered when attempting to build this package: the software requires manual path setup in order to run, the documentation does not build on case-insensitive file systems, and the PDF documentation does not build at all.

⚠1 *Running the built software requires manual path setup*—By default, the `synopsis` build products are placed in a variety of directories:

- `build/clib.linux-x86_64/lib`

- `build/ctemp.linux-x86_64-2.6/src`
- `build/lib.linux-x86_64-2.6/Synopsis`
- `build/temp.linux-86_64/Synopsis/Parsers`
- `build/scripts-2.6`

These directories are used for the libsynopsis shared libraries, intermediate objects files for libsynopsis, Python code, C++ parsers, and command-line program respectively—which requires a lot of manual setup of Python and shared-library-loader search paths in order to run development versions of `synopsis`.

The `README` files describes a custom `setup.cfg` file that can be created to cause the build system to copy all files into one folder, such that the command to run development versions of `synopsis` simplifies to

```
LD_LIBRARY_PATH=$PWD/lib
PYTHONPATH=$PWD/.:$PWD/Synopsis/Parsers/IDL
scripts/synopsis
```

However, the `README` instruction for doing this comes with a warning that it is not to be used with builds that will eventually be installed, because the build system "will get confused and not install extensions."

The remaining two issues involve building the `synopsis` documentation. Such issues might not normally be considered important for building software, but `synopsis` is an unusual instance: it uses itself to scan its own source code to build its own documentation. Building its own documentation is a smoke-test comparable to executing a newly-built GUI application to see if the main application window appears. Additionally, the packaging phase of `synopsis`'s build system includes the documentation in source and binary distributions of `synopsis` if the documentation is present at packaging time, but there is no automatic mechanism to execute the `build_doc` target prior to packaging. Developers must remember to manually execute a documentation build before a release or to get updated documentation.

2 *Documentation does not build on case-insensitive filesystems*—The build system calls a Python script named `synopsis.py` to produce documentation. This script tries to import the module `Synopsis.Parsers.Cxx`. By Python's conventions for importing modules, Python first checks for `Synopsis.py`. On a case-insensitive filesystem—in this case, Mac OS's default HFS+ accessed on Linux via a network share—this results in a case collision and the module is not loaded. Note that there is also a subtle interaction with Python's `.pyc`-file-creating bytecode-caching mechanism that can make this issue hard to reproduce in a standalone example.

This is a portability issue, not only for Mac and Windows machines, but also Linux machines that use Mac and Windows filesystems, such as on network shares.

⚠3 *PDF documentation does not build due to problems in a third-party tool*—The PDF documentation fails to build with an error message from the `xmlroff` DocBook formatting tool: "The QName 'rx:key' has no namespace binding in scope." The error message includes a line number, but there are thousands of XML elements on that line, making it difficult to diagnose the problem. Reformatting the XML file so that there is only one element on each line results in the same error message with the same line number—even though all the line numbers are changed and there is no `rx:key` on the indicated line. This is likely a bug in `xmlroff`.

Since the HTML documentation does build correctly, this is not a serious issue.

What, if any, issues are there in building changes to the package?

There are no issues in building the package, apart from the need to remember to manually execute a documentation build as described above. Python source code is automatically read by the interpreter at run time, so no building is required to incorporate changes. Byte-compilation is performed at packaging time, but this is fast because it is only parsing and serializing source code statements, rather than compiling optimized machine code.

What build system features prevent build problems similar to those encountered in other packages?

① *Automatic dependency extraction from C/C++ source code*—The autotools builds for C/C++ parts use automatic dependency extraction. This helps to handle the dependencies that some parsers have on libsynopsis. If a libsynopsis header file is changed, the automatically extracted dependencies cause the client parsers to be rebuilt. If a libsynopsis implementation file is changed, only the libsynopsis shared library needs to be rebuilt.

② *Shared libraries for subbuilds*—libsynopsis and each parser written in C/C++ are built independently, generating shared libraries. Having distinct shared libraries for each component eliminates build-time delays and dependency problems associated with linking together several components into one library.

③ *Build uses* `distutils` *for its Python core and autotools subbuilds for its C/C++ parts*—This enables mature, well-tested build tools to be used for all parts of `synopsis`. The use of `distutils` as the top-level build with other builds subsidiary to it avoids problems where multiple subbuilds may become out of sync, such as when an API exported from one library is updated and its clients need to be updated appropriately. The top-level `distutils` build automatically ensures that all required sub-builds are executed.

④ *Build system aborts build immediately on error*—in other case studies, the error-handling capabilities of Makefiles and shell scripts are somewhat problematic. In `synopsis`, the complicated parts of the build logic are implemented as Python-language subclasses of objects in the `distutils` API. The build code is able to take advantage of Python's robust error-handling and exception mechanisms. This helps the build system to abort the build immediately when the build issues above are encountered.

Conclusion

`synopsis` uses a top-level `distutils` build with autotools subbuilds for C/C++ components that rely on automatic dependency extraction and shared libraries to help ensure correctness. The only major issues that `synopsis`'s build system has are path configuration issues, and scalability—for five components the overhead of invoking a separate build system for each component totals up to 0.5 seconds for a pure-overhead rebuild when there are no changes. This is fast enough for this system, but such builds would take excessive amounts if they involved hundreds of components.

4.2 Case study: `python3.0`

A summary of the findings for `python3.0` is given in Table 4.2. The multilanguage diagram is Figure 4.5 and the build-time view diagram is Figure 4.7.

What does the package do? What are the major abstractions?

Python is an interactive high-level object-oriented programming language, and `python3.0` is version 3.0 of the interpreter and Standard Library.

Question	Findings
Purpose and abstractions	Interactive high-level object-oriented programming language with functions, classes, methods, and modules
Architecture, languages, and interactions	C for interpreter and builtin objects. C modules interact with Python via equivalent API. Standard Library with ~200 modules, ~80% Python and ~30% C
Build system	autoconf-based with hand-written Makefile, `distutils` for building most of Standard Library
Build issues	None
Rebuild issues	⚠1 Manual dependency management result is that changing some header files forces a full rebuild, while changes to other header files are ignored
Build features	① Checks if running from build directory ② Can extend with Python and C plugins ③ Uses autoconf for configuration ④ Standard `distutils` tool used to build standard library
Conclusion	Few issues

Table 4.2: Summary of findings for `python3.0`

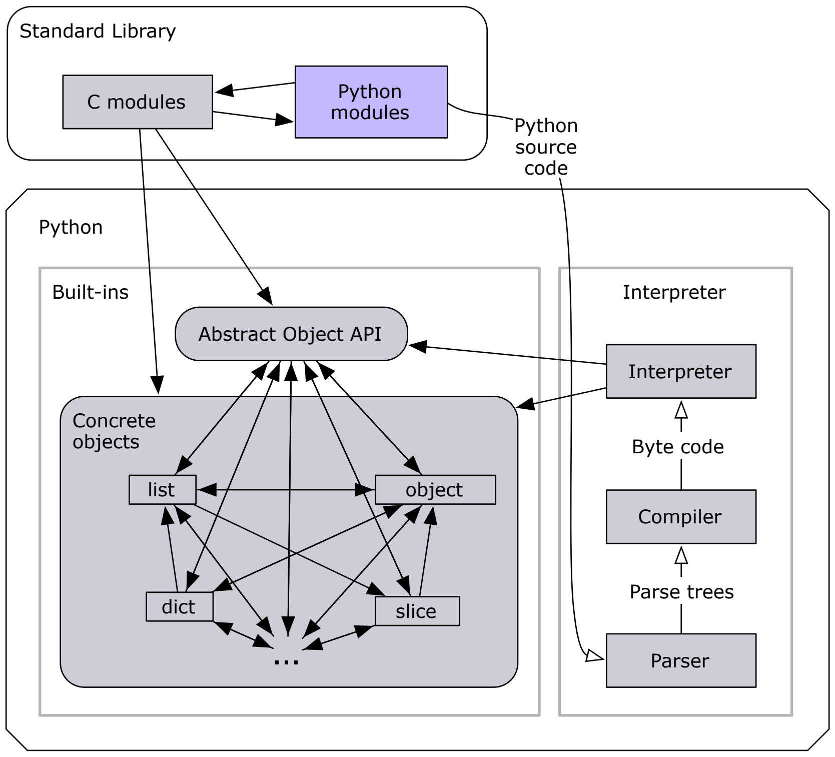


Figure 4.5: `python3.0` multilanguage diagram, using the notation of Figure 4.1. The Standard Library is a collection of interdependent C and Python modules. Python source code is interpreted. C modules bypass the interpreter, but access the same APIs as Python code does. `python3.0`'s built-in objects are implemented in C and are tightly-coupled.

Source code written in Python is organized as a hierarchy of modules containing functions, classes, and methods defined in classes.

What are the major parts of the software, what languages are they written in, and how do they interact?

The major parts of `python3.0` are shown in Figure 4.5. There is an interpreter for the Python programming language; implementations of built-in objects such as `list` that are necessary parts of the language; and a large Standard Library of additional modules. The interpreter executes Python source code by calling the necessary APIs provided by built-in objects. The Standard Library consists of a large collection of useful modules that are not part of the language itself.

The Interpreter

The interpreter is written in C. It executes Python code by parsing Python source code, compiling the Python source code to bytecode, then interpreting the bytecode by calling APIs for built-in objects. As an example of how it works, consider the bytecode execution of the Python statement `return a * b`. The parser parses this statement into the parse tree shown in Figure 4.6.

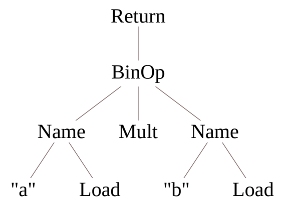


Figure 4.6: `python3.0` abstract syntax tree for `return a * b`

The bytecode compiler compiles this into a small sequence of bytecode, the most relevant fragment of which is the sequence of bytes `0x7C 00 00 7C 01 00 14 53`. `0x7C` is the opcode for `LOAD_FAST`; it is used twice, each time followed by a 16-bit integer in little-endian format. These first six bytes refer to the zeroth and first variables in this context, namely `a` and `b`. `0x14` is the opcode for `BINARY_MULTIPLY` and `0x53` is the opcode for `RETURN_VALUE`. Together these opcodes say, load `a`, load `b`, multiply them, and return the result. When the interpreter processes the `BINARY_MULTIPLY` instruction, it calls the C function `PyNumber_Multiply` defined in the abstract object API to perform the actual multiplication.

The advantage of using bytecode is to simplify the implementation and speed up execution. The implementation is simplified because the parser, compiler, and interpreter are separated with straightforward interfaces between them. Rather than having to deal with parsing and compilation as well, the interpreter is in effect one giant `switch` statement on the opcode. Execution is sped up because, as an optimization, the byte-compiler caches the bytecode on disk when Python source code is compiled. This allows future loads and runs of the same source code to save time by bypassing parsing and compilation.

The Built-Ins

The built-ins provide implementations of objects that are part of the Python language, such as the `list` class. As shown in Figure 4.5, there is a web of tightly-coupled concrete object implementations and an Abstract Object API. The objects are both necessary parts of the language and tightly-coupled because they have been designed together as complementary primitives. For example, the `dict` associative dictionary class is a subclass of the top-level `object` class, and inherits functionality from it, but the namespace of functions and variables associated with an `object` instance is implemented as a `dict`.

The built-in Abstract Object API provides an abstract interface to the concrete objects. This API is roughly equivalent to the semantics of the Python language. For example, at the concrete level, Python `float` and `long` objects are implemented as different C structs with different functions to operate on them. The Abstract Object API provides a `PyNumber_Multiply` function to look up and execute the correct multiplication function, whether it be `float_mul` for `float`s or `long_mul` for `long`s or a custom overloaded multiplication operator for some other class. This is equivalent to simply writing `a * b` in Python source code, and indeed it is `PyNumber_Multiply` that is called by the interpreter when executing such a statement.

The Standard Library

The Standard Library consists of over 200 top-level modules, providing a huge variety of functionality. The modules are written in a combination of C and Python. About 80% of the modules in the Standard Library are written in Python, 30% in C, and 10% in a combination of the two. Python is generally used whenever possible, with C used when low-level system access or extra performance is required. For example, the `termios` module for low-level POSIX TTY I/O services is written entirely in C; the `pprint` module for pretty-printing Python data structures is written entirely in Python; and the `random` module is written in both, with basic uniform random number generation implemented in C, and with transformation to various other probability distributions implemented in Python.

Python modules can be written in C or Python or both, with clients of those modules generally unable to tell the difference. A Python module is a collection of source files. A C module is usually a shared library such as a `.so` or `.dll`, and uses the same API as the concrete built-ins. How can one module be both a collection of Python source files and a binary shared library? The answer is a consequence of the fact that all interaction in Python is done via object-oriented interfaces: Python modules via the semantics of the Python language, and C modules via the same APIs used by the Python interpreter. Code in either language can operate on any Python objects in equivalent ways without regard for the target object's implementation language. Python and C modules can arbitrarily access, subclass, and import from other modules without regard for the implementation language of the target. A module written in both C and Python is actually two modules, say `mymodule` in Python and `_mymodule` in C. But since classes defined in one can be extended in the other, and methods in either language call each other the same way, this separation of languages is merely an implementation detail hidden from callers.

What is the structure of the build system and how is it implemented?

The structure of `python3.0`'s build is shown in Figure 4.7. First, an executable interpreter capable of running `distutils` is built. This requires that certain Standard Library C modules needed by `distutils`, such as the `_sre` module implementing regular expressions, be compiled into the `python3.0` interpreter. Once the interpreter is built, all Python modules in the Standard Library can be used. They are simply copied to the execution environment. Finally, `distutils` compiles the remaining C modules of the Standard Library into indi-

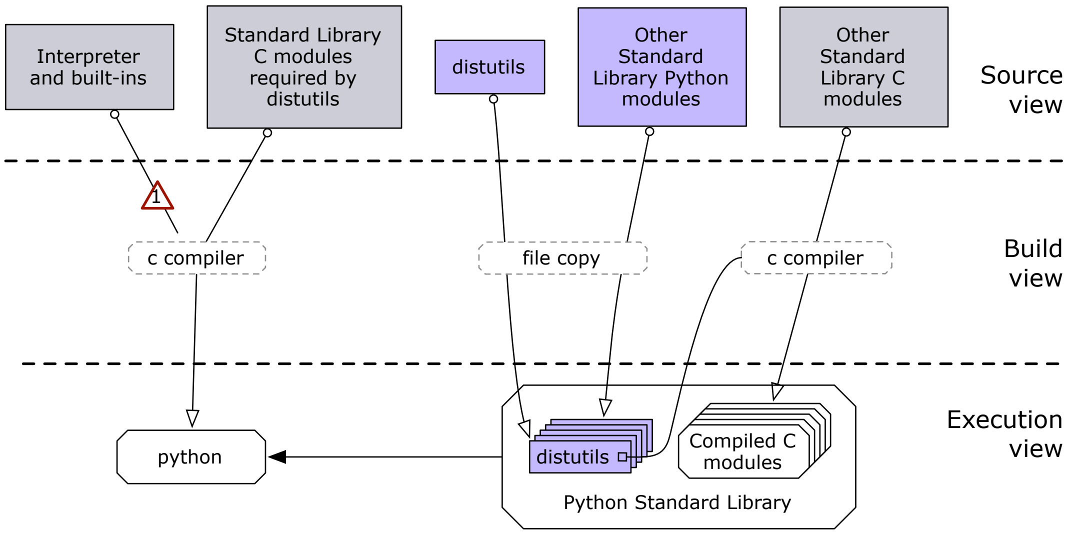

Interpreter and built-ins
Standard Library C modules required by distutils
distutils
Other Standard Library Python modules
Other Standard Library C modules
Source view
1
c compiler
file copy
c compiler
Build view
python
distutils
Compiled C modules
Python Standard Library
Execution view

Figure 4.7: `python3.0` build diagram, using the notation of Figure 4.1. The interpreter, built-in objects, and the C portion of the Standard Library required by `distutils` are compiled into an executable binary. This allows `distutils` to run, and it compiles the remainder of the Standard Library.

vidual shared libraries. `distutils` also manages the configuration of these modules. For example, certain optional database client modules in the Standard Library are built only if the necessary libraries and header files are available.

The building of the interpreter uses autoconf and Make. autoconf is used for its configuration phase. For example it checks which threading libraries are available and work correctly. The construction phase uses a handwritten Makefile of approximately 1200 lines. It is somewhat shorter and more comprehensible than typical autoconf-generated Makefiles.

What, if any, issues are there in building the package?

No build issues were encountered with `python3.0`. It simply worked. The configure script and Makefile executed quickly, producing a functioning interpreter, which was able to run `distutils` to build the Standard Library.

What, if any, issues are there in building changes to the package?

There was, however, one issue with rebuilds.

⚠1 *Incorrect manually-specified dependencies result in full rebuilds for some source changes, while some other source changes are ignored*—instead of using an automated tool to extract dependencies between intermediate object files and C source and header files, `python3.0`'s handwritten Makefile contains manually-specified dependencies. The relevant Makefile fragment is shown in Figure 4.8. It declares that all intermediate object files depend on all header files. The result is that changing any exported header file forces a full rebuild of all source

```
...
PYTHON_OBJS=        \
                Python/_warnings.o \
                Python/Python-ast.o \
                Python/asdl.o \
                Python/ast.o \
                Python/bltinmodule.o \
                Python/ceval.o \
                Python/compile.o \
                ... at least 33 more object files ...
                ... (depending on configuration) ...
...
PYTHON_HEADERS= \
                Include/Python-ast.h \
                Include/Python.h \
                Include/abstract.h \
                Include/asdl.h \
                ... 74 more header files ...
...
LIBRARY_OBJS=        ... $(PYTHON_OBJS) ...
...
$(LIBRARY_OBJS) $(MODOBJS) Modules/python.o: $(PYTHON_HEADERS)
...
```

Figure 4.8: Python Makefile fragment

files, whether or not any particular source file included the changed header file or not. As well, these manually-maintained lists of dependencies can easily get out of synchronization with the source code, and in fact some of them are—`Python/importdl.h` is not mentioned in any dependency lists, yet is included by several source files. Changes to this header file are ignored by the build system.

In contrast, a typical autoconf-generated Makefile, or one calling on a tool such as makedepend—as `synopsis`'s Makefiles for C/C++ components do—automatically extracts the correct dependencies from the source code at build time.

What build system features prevent build problems similar to those encountered in other packages?

① *Checks if running from build directory*—`python3.0` runs without issue from inside the build directory. `python3.0` sidesteps the path issues encountered with `synopsis` by checking at startup inside `getpath.c` whether it is running from a build directory, and, if so, adjusts its library search paths accordingly.

② *Can extend with Python and C plugins*—`python3.0` is structured as a C-language core which modules written in Python or C can interact with through equivalent APIs. Although there are no problems with building `python3.0`, most developers adding functionality to `python3.0` by contributing new libraries can do so without ever building `python3.0` at all.

③ *Uses autoconf for configuration*—`python3.0` uses the standard autotools tool suite to determine a configuration in a separate phase prior to construction. This avoids problems experienced in the `axiom` case study to come, where configuration is transiently interleaved with construction.

④ `distutils` *tool used to build standard library*—The `distutils` tool used for building the Standard Library is also the standard tool for building third-party Python and C modules. This ensures that `python3.0`-project knowledge about how to build code using the `python3.0` APIs is captured in a reusable, general-purpose manner.

Conclusion

No issues were encountered in building `python3.0`. A single handwritten Makefile builds the approximately 100 KSLOC C "core." However, the manual dependency management used in this Makefile can be error-prone, and can result in lengthy and broken rebuilds.

4.3 Case study: `gnat-gps`

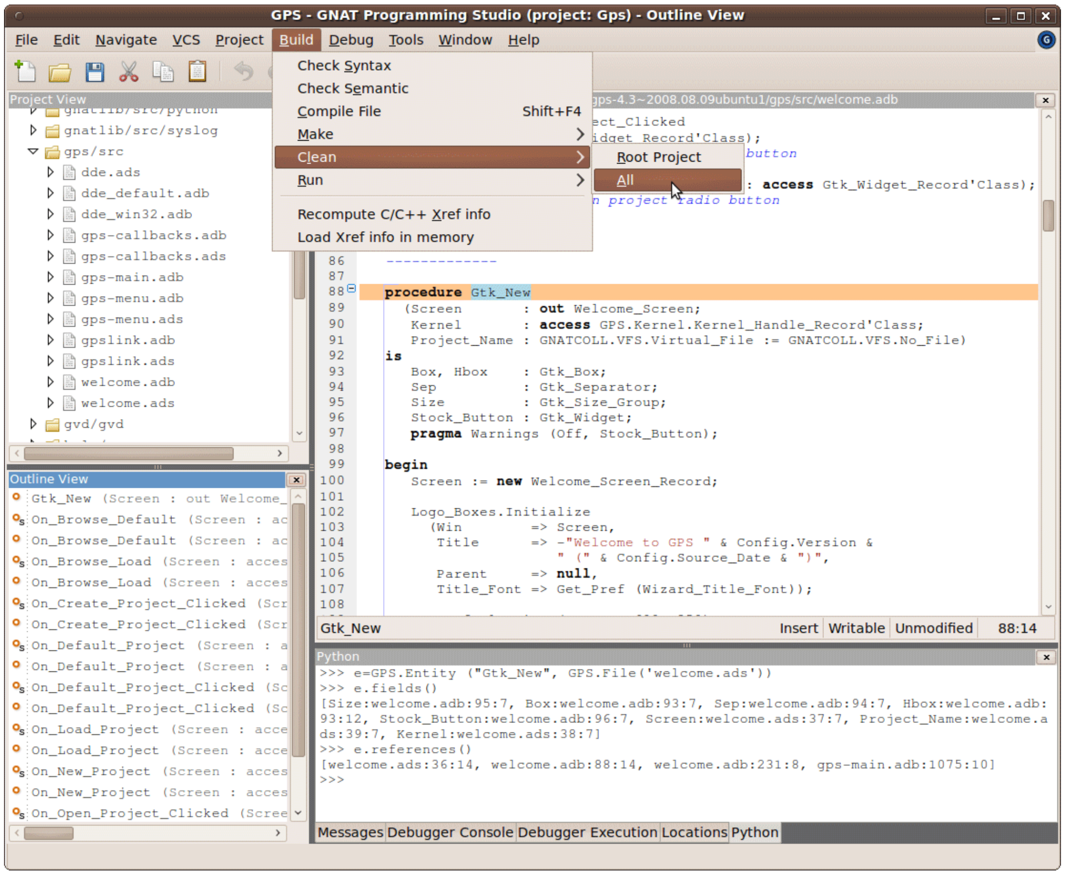
A summary of the findings for `gnat-gps` is given in Table 4.3. The multilanguage diagram is Figure 4.10. `gnat-gps` has two build systems. The build-time view diagram for the "AdaCore" build system is Figure 4.11, and the build-time view diagram for the "Brenta" build system is Figure 4.12.

What does the package do? What are the major abstractions?

`gnat-gps`, also known as GNAT-GPS or GNAT Programming Studio, is an interactive development environment for Ada [70] and C/C++ source code. It provides features such

Question	Findings
Purpose and abstractions	Integrated Development Environment for Ada and C/C++
Architecture, languages, and interactions	Primarily object-oriented Ada architecture, with Python plug-ins, and C support libraries and tools
Build system	Combination of Make and the Ada-specific `gnatmake` tool
AdaCore build issues	⚠1 References Ada libraries no longer part of the project ⚠2 Omits search paths for many third-party C libraries ⚠3 Incompatible project file syntax ⚠4 Inconsistencies between Ada and system interfaces ⚠5 Ubuntu maintainer rewrites build system
Brenta build issues	⚠6 Blank icons, dialogs when run from build directory ⚠7 Python console does not show output
Rebuild issues	No issues
Build features	① Scripting interface allows build-free extension ② `gnatmake` handles dependencies automatically ③ Build aborts immediately on error
Conclusion	Vendor build system hard to build, but rewritten build system also has issues

Table 4.3: Summary of findings for `gnat-gps`


GPS - GNAT Programming Studio (project: Gps) - Outline View
File Edit Navigate VCS Project Build Debug Tools Window Help
Check Syntax
Check Semantic
Compile File Shift+F4
Make
Clean
Run
Recompute C/C++ Xref info
Load Xref info in memory
Root Project
All
Project View
gnatlib/src/syslog
gps/src
dde.ads
dde_default.adb
dde_win32.adb
gps-callbacks.adb
gps-callbacks.ads
gps-main.adb
gps-menu.adb
gps-menu.ads
gpslink.adb
gpslink.ads
welcome.adb
welcome.ads
gvd/gvd
Outline View
Gtk_New (Screen : out Welcome_
On_Browse_Default (Screen : ac
On_Browse_Default (Screen : ac
On_Browse_Load (Screen : acces
On_Browse_Load (Screen : acces
On_Create_Project_Clicked (Scr
On_Create_Project_Clicked (Scr
On_Default_Project (Screen : a
On_Default_Project (Screen : a
On_Default_Project_Clicked (Sc
On_Default_Project_Clicked (Sc
On_Load_Project (Screen : acce
On_Load_Project (Screen : acce
On_New_Project (Screen : acces
On_New_Project (Screen : acces
On_Open_Project_Clicked (Scree
gps-4.3~2008.08.09ubuntu1/gps/src/welcome.adb
Gtk_New
Insert
Writable
Unmodified
88:14
Python
Messages
Debugger Console
Debugger Execution
Locations
Python

```
   --------------

   procedure Gtk_New
     (Screen       : out Welcome_Screen;
      Kernel       : access GPS.Kernel.Kernel_Handle_Record'Class;
      Project_Name : GNATCOLL.VFS.Virtual_File := GNATCOLL.VFS.No_File)
   is
      Box, Hbox    : Gtk_Box;
      Sep          : Gtk_Separator;
      Size         : Gtk_Size_Group;
      Stock_Button : Gtk_Widget;
      pragma Warnings (Off, Stock_Button);

   begin
      Screen := new Welcome_Screen_Record;

      Logo_Boxes.Initialize
        (Win        => Screen,
         Title      => -"Welcome to GPS " & Config.Version &
                       " (" & Config.Source_Date & ")",
         Parent     => null,
         Title_Font => Get_Pref (Wizard_Title_Font));
```

```
>>> e=GPS.Entity ("Gtk_New", GPS.File('welcome.ads'))
>>> e.fields()
[Size:welcome.adb:95:7, Box:welcome.adb:93:7, Sep:welcome.adb:94:7, Hbox:welcome.adb:
93:12, Stock_Button:welcome.adb:96:7, Screen:welcome.ads:37:7, Project_Name:welcome.a
ds:39:7, Kernel:welcome.ads:38:7]
>>> e.references()
[welcome.ads:36:14, welcome.adb:88:14, welcome.adb:231:8, gps-main.adb:1075:10]
>>>
```

Figure 4.9: `gnat-gps` screenshot

as syntax highlighting, code completion, call graph browsers, and a debugger. Although it also supports editing C/C++ source, it is primarily intended for Ada development. A screenshot is shown in Figure 4.9.

`gnat-gps` is developed by AdaCore, the commercial Ada vendor which maintains GNAT, the open-source Ada compiler that is part of GCC [56]. AdaCore also uses `gnat-gps` as a different sort of development environment, namely as an incubator for Ada libraries. Many `gnat-gps` features are implemented in general-purpose ways, then broken off and released as separate libraries once they are mature. These libraries can be used by any Ada program, making Ada a more attractive language for development. Examples of libraries developed as part of `gnat-gps` include GtkAda, the bindings library that `gnat-gps` uses to access the native windowing system from Ada code; XmlAda, for XML processing; and GNATCOLL, the GNAT Component Collection that provides functionality such as scripting language interfacing, logging, and file manipulation.

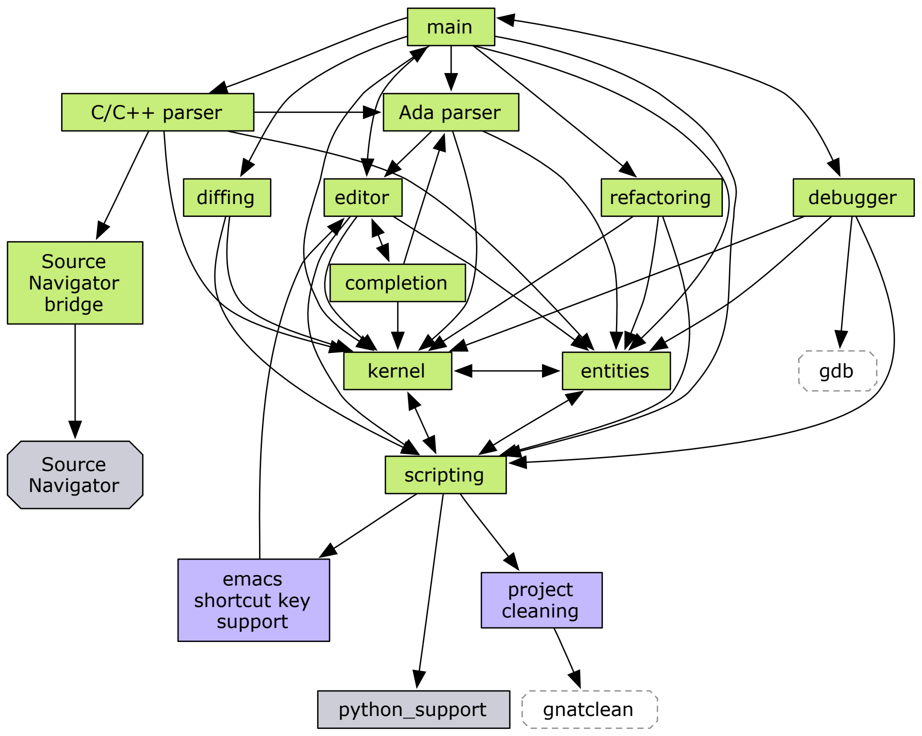


Figure 4.10: `gnat-gps` multilanguage diagram, using the notation of Figure 4.1. Only a sample of `gnat-gps` modules are shown. The core is a large collection of Ada modules of related objects. Modules that implement high-level feature are loaded by the `main` module. Most modules accomplish their tasks by calling lower-level modules and interacting with the `kernel` module. The `entities` module manages semantic data about source code entities. Many modules register themselves with the scripting interface, which loads features written in Python. Source Navigator is invoked as a separate executable.

What are the major parts of the software, what languages are they written in, and how do they interact?

`gnat-gps` is mainly composed of a network of interacting Ada objects, with Python plug-ins to implement and expose other features, and C support code. The use of a central "kernel" module is reminiscent of blackboard architectures [25]. The multilanguage diagram of a selected portion of `gnat-gps` is shown in Figure 4.10.

The majority of `gnat-gps` is implemented as a collection of Ada objects interacting in an object-oriented architectural style [25]. Ada is an object-oriented programming language with an emphasis on developing reusable components. For example, instead of implementing its own parser, `gnat-gps`'s support for parsing and interpreting the `gnatmake` project files described below is enabled by using the `gnatprj` library. The `gnatprj` library is a key part of the `gnatmake` build tool, and is distributed with the GNAT compiler.

Nearly all of the C source code in `gnat-gps`—97% of it—is a modified version of Red Hat's open-source Source Navigator project. It provides fact extraction and cross referencing for C/C++. `gnat-gps` executes Source Navigator binaries to extract facts from source code, and then `gnat-gps` reads from the data files written out by Source Navigator. As it runs in a separate process from `gnat-gps` and is not integrated with the main `gnat-gps` IDE, Source-Navigator is not considered part of `gnat-gps` proper.

The remaining `gnat-gps` C code consists of pixmaps and utility code. Pixmaps are described in Figure 3.4. The utility code exposes low-level functionality such as `tty` manipulation, memory mapping, and determining the path of the system `temp` folder in a cross-platform way. Although Ada makes calling external C functions trivial—one simply declares the function and issues a `pragma Import` statement—C code is still used for invoking functions with complicated signatures involving macros, constants, and struct layouts. For example, the Windows API for displaying a native file selection dialog is a macro alias for a function with 20 parameters, most of which are themselves structs and/or combinations of constants defined in Windows header files. Correctly defining all the required constants, macros, and structs in Ada is much more difficult than writing a simplified wrapper function in a C file that includes windows header files, and calling the wrapper from Ada. Similarly, `gnat-gps` calls most Python API functions directly from Ada, but provides C-language wrappers for Python API functions that are actually defined as macros.

There are Python bindings for the complete `gnat-gps` object model. There is also an interactive console, as shown in Figure 4.9, where the references to the current unit are being queried. The mechanism that implements the bindings is shown schematically in Figure 4.10. Modules throughout the package register their functionality with the scripting interface manager. The scripting interface manager invokes scripts and enables the scripts to call back into `gnat-gps`. The scripting interface is abstract, with two concrete implementations: one for Python, and one for an unnamed `gnat-gps`-specific shell-like language.

Many key features of `gnat-gps` are implemented in Python. For example, the Build → Clean → All menu shown in Figure 4.9 is implemented as a Python script that extends the user interface and calls the external `gnatclean` compiler tool to do the actual work. Users can also extend the interface by providing their own Python scripts with access to the same API. Users can develop their scripts interactively using the Python console, then place them in the special directory `~/.gps/plugins` to be automatically loaded by `gnat-gps`. `gnat-gps` comes with a 60-line example script that adds a "soft tabs" feature to the IDEs source code editor, whereby pressing the tab key inserts the syntactically correct amount of indentation rather than a literal tab character.

What is the structure of the build system and how is it implemented?

Before describing the build system for `gnat-gps`, it is first necessary to describe some features of the GNAT compiler for Ada.

Background

Unlike other Ada compilers which use a database-like master "library file" to manage compiled code, GNAT uses a C-like compilation model where implementation files are compiled to individual intermediate object files which can then be linked together [16, 26]. To ensure consistency, at compilation time the compiler writes out an Ada Library Information `.ali` file containing the checksums of all referenced source files along with tool versions and compiler flags. At link time, a tool called the binder verifies consistency before calling the linker. This avoids "version skew" problems in which objects compiled against different versions of other objects are linked together.

The `gnatmake` build tool included with the GNAT compiler is often used to build Ada software. It reads in `.gpr` GNAT project files declaring what is to be built, and, using the same `.ali` files and parsing routines as the compiler and linker, rebuilds those files which are missing or inconsistent. The syntax for project files is extremely similar to Ada source code. `gnatmake` supports hierarchical builds using the same concepts, such as `with` statements, used by Ada for inheritance. There is some support for configuration-phase functionality in the form of conditionals that can appear inside project files. Configuration can also be performed using tools such as autoconf to parameterize Ada-language source and project files.

`gnat-gps` build systems

As is discussed below, there are two build system for `gnat-gps`: the original, vendor-provided "AdaCore" build system, and a replacement "Brenta" build system written by Ludovic Brenta, the Ubuntu maintainer for the `gnat-gps` package. Each uses a combination of Makefiles to build C code and `gnatmake` to build Ada code. Diagrams of each are shown in Figure 4.11 and Figure 4.12. The key difference is that the Brenta system builds the entire package at once, while the AdaCore system builds each source directory independently.

The *AdaCore build system* is a hierarchy of cross-platform Makefiles that uses a Make-based build framework developed by AdaCore for C, C++, and Ada. Each source directory contains both a Makefile and a `.gpr` GNAT project file for building that directory and its prerequisite directories. The Makefile in each source code directory textually includes the Makefiles for the source directories it depends on. The structure of Makefile includes is shown in the Source View graph of Figure 4.11. Intermediate object files are built in each source directory, then linked together at the end to create the `gnat-gps` executable.

The *Brenta build system* uses a single Makefile and a single `.gpr` GNAT project file for all `gnat-gps` source code. The structure of the build system is shown in Figure 4.12. There is a single Makefile for all C code, and a single project file for all Ada code. The entire project is compiled and linked together. The configuration of this build system is

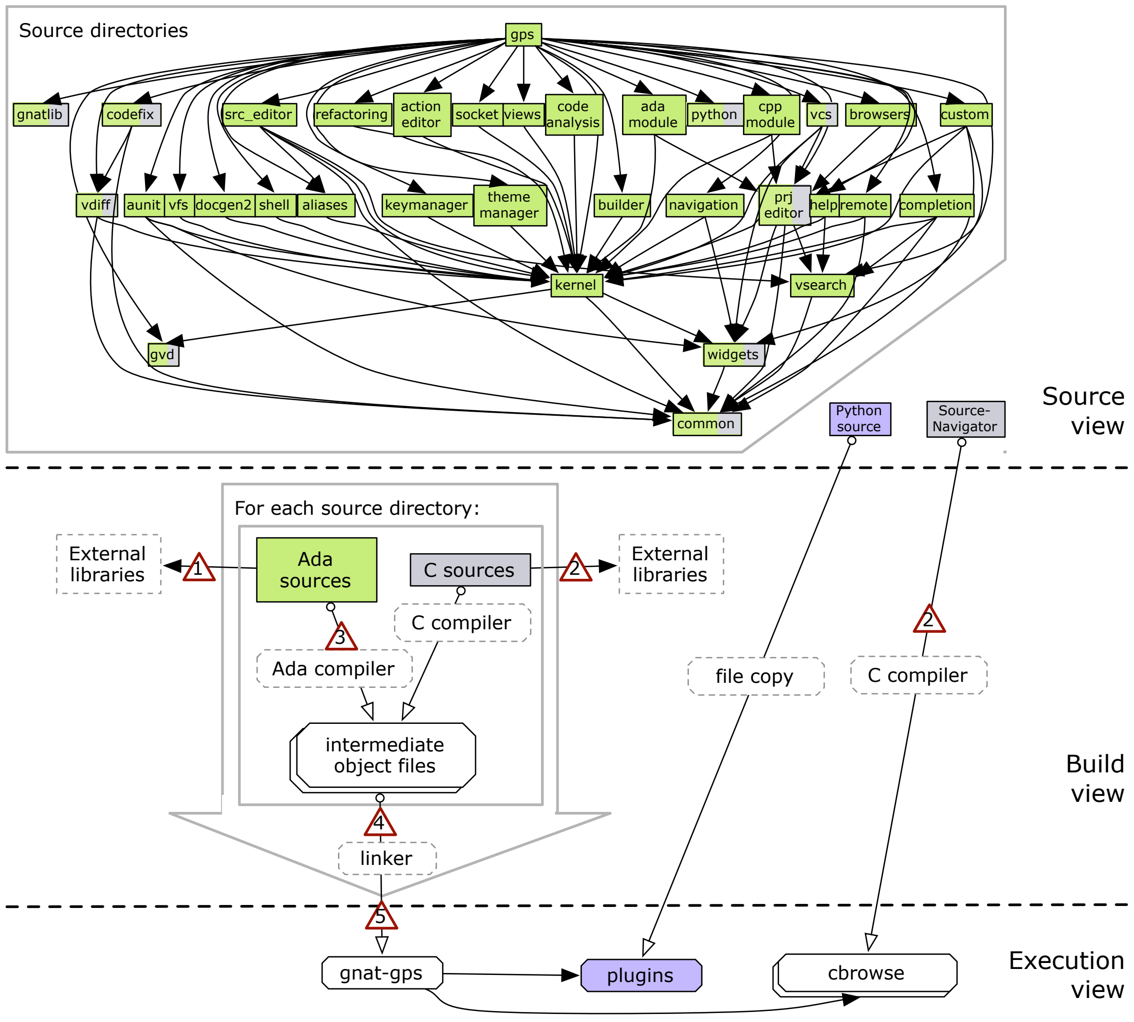


Figure 4.11: `gnat-gps` build diagram, AdaCore build system, using the notation of Figure 4.1. Each source code directory has its own build system that invokes the build systems of all directories containing source code that it uses. Inside each directory, Ada and C sources are compiled to intermediate object files. Once all directories are built, the object files are linked together to form the `gnat-gps` executable.

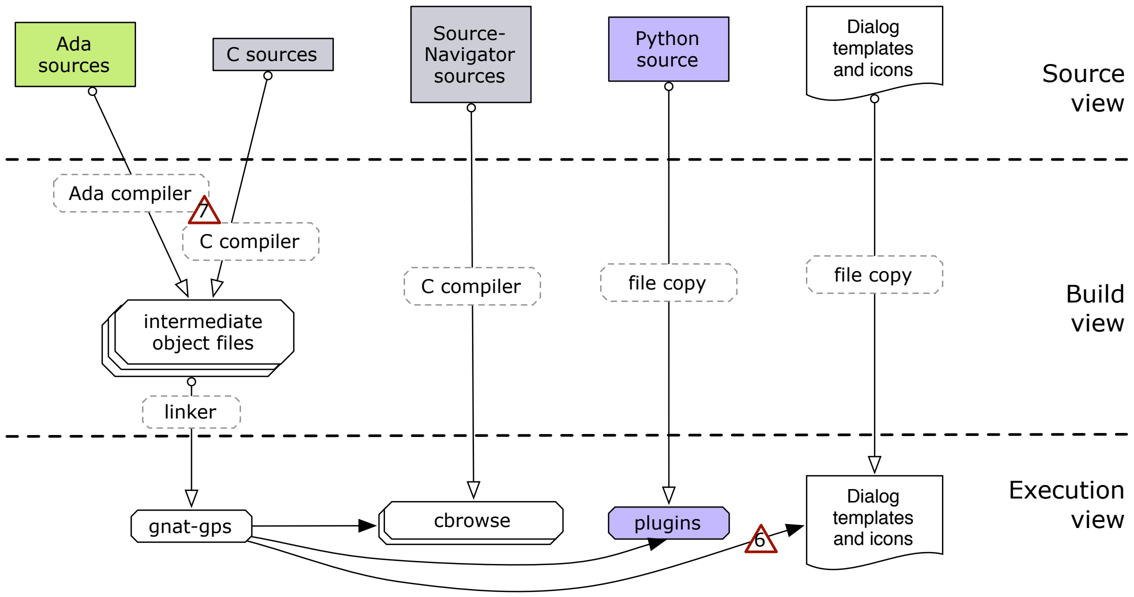


Figure 4.12: `gnat-gps` build diagram, Brenta build system, using the notation of Figure 4.1. All `gnat-gps` Ada sources are compiled to intermediate object files; all C sources are compiled to intermediate object files; and the resulting set of object files is linked together to create the `gnat-gps` executable.

hard-coded with Ubuntu-specific values. Brenta compares his build system to the AdaCore system in a source code comment:

> “This project file bypasses the complex structure of project files importing and including each other, brittle configure scripts, evil recursive Makefiles and Makefile fragments supplied by AdaCore. Now there is just one Makefile (`debian/rules`), and one project file, which you are now reading. This radical simplification is the Tao of Programming. In this file, I do away with all the flexibility and configurability of upstream’s build machinery. I don’t need them. I only care about Debian.
>
> “Now, if you go to the top level of the directory and just call debian/rules with no arguments, you get The Right Thing. Similarly calling "gnatmake -Pgps" does The Right Thing for the Ada part of GPS.”
>
> —Ludovic Brenta, `gps.gpr`, 2006

What, if any, issues are there in building the package using the AdaCore build system?

Discussion of the following build issues is complicated by the fact that the Ubuntu-supplied source code for `gnat-gps` differs from the original vendor source code. As mentioned on page 13, 1.6% of Ubuntu packages have modified source code packages, while the remaining

98.4% distribute the original vendor source code alongside a Ubuntu patch file. Different variants of the source code of the same version `gnat-gps` can be obtained from several locations: the Ubuntu-modified version from the Ubuntu archives;[1] AdaCore's publically-available source-control server;[2] and source code archives that may be obtained by navigating AdaCore's interactive Download Package Builder.[3] As discussed below as Build Issue ⚠1 , `gnat-gps` relies on external libraries that were once part of `gnat-gps`, and the AdaCore build system expects the source code for those external libraries to be in the same place as they once were. The Download Package version is the most complete, but cannot be automatically retrieved from a stable URL. The version from the source-control server is identical but only includes source code that is strictly part of `gnat-gps`. The Ubuntu version falls between these—it is not complete but does include some source code that is not strictly part of `gnat-gps`—but also adds its own modifications. Finally, the Download Package version of `gnat-gps` appears to be a slightly different version from what the Ubuntu archive with the same version number is based on.

In short, there are many variants of the source code for this package, among which the following problems were encountered. Not every variant necessarily exhibited every one of these build problems, but build problems ⚠1 and ⚠3 were universal across all variants, and no variant of the software succeeded in building using the AdaCore build system.

⚠1 *There are references to Ada libraries which are no longer part of the project*—the Ada libraries templates_parser, GtkAda, and XmlAda were once part of `gnat-gps` but have since become independent libraries that are distributed separately. For example, templates_parser is available in Ubuntu as the independent package `libtemplates-parser11`, which is used by both `gnat-gps` and the Ada Web Server framework project. However, inside the `gnat-gps` source code, the AdaCore build system still expects a `templates_parser` source directory that can be built as part of `gnat-gps`. As this is missing from the Ubuntu-distributed source code, build failures occur. Conversely, the `gnatlib` directory contains a library that is still part of the source code of `gnat-gps`, but which has its own `configure` script as if it were an independently-built library; this configure script is not automatically invoked, causing build failures. Additional build failures also occur due to the build system expecting the source code for Tcl/Tk and the GNAT compiler to be present, despite their not being distributed with the `gnat-gps` source code.

This would be better handled if the configuration script checked for both system-installed and local source versions of required libraries. Local source versions may still be desired when development is occurring to fix bugs or to add new library features to support `gnat-gps`.

[1] http://old-releases.ubuntu.com/ubuntu/pool/universe/g/gnat-gps/gnat-gps_4.3~2008.08.09ubuntu1.tar.gz

[2] http://svn.eu.adacore.com/anonsvn/Dev/trunk/gps @r127639

[3] http://libre.adacore.com/download/configurations

```
 1  with "../kernel/kernel";
 2  with "../shared";
 3
 4  project Help is
 5
 6     for Source_Dirs use ("src");
 7     for Object_Dir use "obj";
 8
 9     package Naming is
10        case Shared.OS is
11           when "Windows_NT" =>
12              for Implementation ("help_module.display_help")
13                use "help_module-display_help_win32.adb";
14           when "unix" =>
15              for Implementation ("help_module.display_help")
16                use "help_module-display_help_unix.adb";
17        end case;
18     end Naming;
19
20     package Compiler renames Shared.Compiler;
21     package IDE renames Shared.IDE;
22
23  end Help;
```

Figure 4.13: `help.gpr` project file

⚠2 *Search paths for third-party C libraries are omitted*—`gnat-gps` relies on C header files and shared libraries for a number of third-party packages, namely Python and the GUI libraries gtk, atk, glib, pango, and cairo. However, while the configure script does search for these libraries and determine the include file search paths for them, this information does not propagate to every compilation that requires them, and build failures result.

In the builds where this is encountered, it is possible to add them manually. Doing this for six different prerequisite packages can become tedious.

⚠3 `gnatmake` *fails to parse conditionals in project files*—While the AdaCore build system does use an autotools-generated `configure` script, some configuration decisions are handled at the `gnatmake` project file level by conditionals. For example, the highlighted fragment in Figure 4.13 specifies that the `display_help` interface has an OS-specific implementation body. `gnatmake` fails to parse this, yielding the error messages `variable "os" is not typed`, `unknown variable "Os"`, and `illegal case label`.

The syntax for the conditional is as specified in the GNAT manual [1]. The apparent cause is a difficulty in accessing the `shared` project file referenced on line 2. It is also referenced by the `kernel` project file referenced on line 1, which in turn references many other project files. A warning about a `duplicate project name "shared"` appears eleven times with the error message about the case level. This confusion is likely an aspect of what Brenta refers to as the "complex structure of project files importing and including each other" in the source code for his replacement build system.

One possible workaround for this build issue is to manually resolve and remove the conditionals from all project files in which they cause errors; while this does allow the build to proceed, it is not necessarily an effective workaround, as the build still does not ultimately succeed.

⚠4 *There are inconsistencies between Ada and system interfaces*—`gnat-gps` calls various C libraries, either directly from its own source code or indirectly via binding libraries such as GtkAda. The underlying C libraries can change in incompatible ways as new versions are released. One version of GtkAda refers to the struct fields `n_bytes` and `text_size` in the Gtk API. However, these fields have been renamed to `x_n_bytes` and `x_text_size` in Gtk. This error is caught by the linker.

It is possible to get `gnat-gps`'s compilation to proceed past this issue by renaming the Ada declarations of the field as well. However, it is likely that there was a reason that these fields were changed to names conveying the idea that they have been removed or "`x`"ed out, and the GtkAda library may need further source code changes to adapt to the API change.

⚠5 *This build system is sufficiently hard to get working that the Ubuntu maintainer wrote a new build system*—several other serious problems were encountered in trying to build this system. One such issue was the persistent inability of the compiler to find the `g-traces.ads` file needed throughout the system, even though it exists in what seems to be the correct directory. Another was a build that appeared to proceed into the linking stage only for the link to fail with an error message that all objects needed to be recompiled as Position-Independent Code by using the `-fPIC` compiler option.

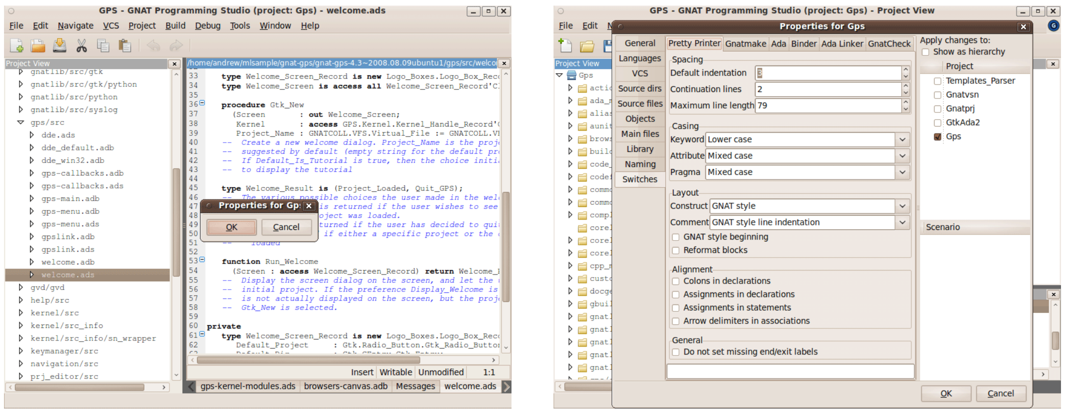
After encountering so many problems, many of which could not be easily solved, the Ubuntu modifications to the build system were examined to see how it managed to build successfully. Surprisingly, instead of modifying the build system to avoid build issues, it was rewritten completely.

What, if any, issues are there in building the package using the Brenta build system?

The Brenta build system is designed to build system-installable binary packages for Ubuntu. There is no configuration phase because values for the desired configuration are hard-coded into the build system. It is also not without problems.

⚠1 `gnat-gps` *requires installation to run properly*—while `gnat-gps` does build and run on the first try, the resulting application is not functional until installed. Figure 4.14 shows the main `gnat-gps` binary running from the development version on the left, and the installed version on the right. The development version is missing the folder icons in the Project View on the left, and, more importantly, the extensive Project Properties configuration dialog shown on the right is replaced by a blank dialog featuring only Ok and Cancel buttons.

The cause is that resources such as icon files and also the XML templates used to render



```
   type Welcome_Screen_Record is new Logo_Boxes.Logo_Box_Rec
   type Welcome_Screen is access all Welcome_Screen_Record'Cl

   procedure Gtk_New
     (Screen       : out Welcome_Screen;
      Kernel       : access GPS.Kernel.Kernel_Handle_Record'(
      Project_Name : GNATCOLL.VFS.Virtual_File := GNATCOLL.VF
   --  Create a new welcome dialog. Project_Name is the proj
   --  suggested by default (empty string for the default pr
   --  If Default_Is_Tutorial is true, then the choice initi
   --  to display the tutorial

   type Welcome_Result is (Project_Loaded, Quit_GPS);
   --  The various possible choices the user made in the wel
                    is returned if the user wishes to see
                    oject was loaded.
                    turned if the user has decided to qui
                      if either a specific project or the
   --     loaded

   function Run_Welcome
     (Screen : access Welcome_Screen_Record) return Welcome_I
   --  Display the screen dialog on the screen, and let the
   --  initial project. If the preference Display_Welcome is
   --  is not actually displayed on the screen, but the proj
   --  Gtk_New is selected.

private
   type Welcome_Screen_Record is new Logo_Boxes.Logo_Box_Rec
      Default_Project     : Gtk.Radio_Button.Gtk_Radio_Butto
```

General | Languages | VCS | Source dirs | Source files | Objects | Main files | Library | Naming | Switches

Pretty Printer | Gnatmake | Ada | Binder | Ada Linker | GnatCheck

Spacing
- Default indentation: 3
- Continuation lines: 2
- Maximum line length: 79

Casing
- Keyword: Lower case
- Attribute: Mixed case
- Pragma: Mixed case

Layout
- Construct: GNAT style
- Comment: GNAT style line indentation
- ☐ GNAT style beginning
- ☐ Reformat blocks

Alignment
- ☐ Colons in declarations
- ☐ Assignments in declarations
- ☐ Assignments in statements
- ☐ Arrow delimiters in associations

General
- ☐ Do not set missing end/exit labels

Apply changes to:
- ☐ Show as hierarchy

Project
- ☐ Templates_Parser
- ☐ Gnatvsn
- ☐ Gnatprj
- ☐ GtkAda2
- ☑ Gps

Scenario

Figure 4.14: `gnat-gps`, development vs. installed versions

some dialogs are supposed to be in the system data directory `/usr/share/gps`, but this does not exist until `gnat-gps` is installed. These resources are scattered across many different source directories and gathered together at install time into a different directory structure.

If another version of `gnat-gps` is already installed, the resources from that version will be used, but these may be out of synchronization, especially if, for example, a developer is working on the Project Properties dialog. The AdaCore build system is able to gather resource files into a temporary directory that `gnat-gps` can be configured to use. The Brenta build system also gathers resource files into a temporary directory while building a binary package, but it automatically deletes that same directory as part of the process, making the temporary directory unusable as a development resource path.

2 *The Python console does not work due to inconsistent configuration*—`gnat-gps` includes an interactive Python console, as seen in the lower-right of Figure 4.9. However, in the built version of `gnat-gps`, the output of Python statements does not appear on the console. The console is interpreting Python statements, as can be tested by executing a Python command to, say, create a file. However, no output appears.

After a great deal of debugging, it turns out that the version of the Python library used at link-time is hardcoded in the `.gpr` project file as version 2.5, while the Makefile that compiles Python-supporting C source code sets the header file search path by querying the operating system for its default python version, namely 2.6. One result of compiling against a different version of the headers from the library version used for linking is that the `PyString_Check` call in the Python API always returns false. It is supposed to return whether its argument is of string type; it is actually a macro that inspects Python-private data structures that are different between the two Python versions. Since it always returns false, the Ada code that calls it believes that the string objects coming from Python to be printed in the console are not strings, so, instead of printing the output string, it raises an exception which is logged but not reported to the end-user. The exception message—

“`Parameter 1 should be a string`”—is of limited usefulness in debugging the issue. Presumably the malfunctioning Python type-checking calls cause undesirable behaviour in other parts of `gnat-gps` as well.

Modifying the Makefile to be consistent with the `.gpr` project file restores correct behaviour. It is likely that, at the time the build system was written, version 2.5 was the default version of Python, so that the build was consistent, but the default changed between then and when this version of Ubuntu was released.

What, if any, issues are there in building changes to the package?

Once `gnat-gps` was built, no issues were encountered in rebuilding after making changes to the source code. Multiple files were repeatedly modified over an extended period of time while debugging the Python-console issue. Each time, the build system quickly produced an updated executable. When no files are changed, it takes about 2 seconds for the build system to determine that no further compilation is necessary. When one or two source files are changed, it take about 8 seconds to rebuild that one file and link the 63MB binary.

One interesting feature of `gnatmake` is the ability to ignore whitespace and comments when deciding whether file are up-to-date. `gnatmake` calls the GNAT compiler's object-oriented input routines when performing its checksumming of source code. This provides it with an option to ignore whitespace and comments when checksumming, eliminating rebuilds when only comments or whitespace have changed. However, if lines are inserted or deleted, then line numbers in debugging information generated at compile-time may be slightly incorrect.

What build system features prevent build problems similar to those encountered in other packages?

① *Scripting interface allows build-free extension*—`gnat-gps`'s interactive scripting interface allows full access to `gnat-gps`'s object model, and allows it to be customized and new features to be developed for it using scripting languages instead of compiled Ada source code. This allows the build system all of its possible issues to be completely avoided for many types of development tasks. In fact, since some critical features of `gnat-gps` are implemented using the scripting interface, it may be that the `gnat-gps` developers prefer the scripting interface over compiled Ada source code.

② `gnatmake` *handles dependencies automatically*—the `gnatmake` tool automatically extracts header-file dependencies from Ada source code. This prevents incorrect builds arising from changes to header files being missed by the build systems. Some dependency extraction tools for C source code may not always be exact, due to differences between the extraction tool and the compiler in terms of header file search paths, defined preprocessor variables, or nonstandard preprocessor syntax extensions. However, when the **gnatmake**-based build system performs dependency extraction directly from the build tool that also invokes the compiler, and calls the compiler's API directly, dependencies are exactly those as seen by the compiler. In fact, **gnatmake**-based build systems have sufficiently detailed knowledge about the build and the compilation that they can correctly handle inlining implementation code into other build units.

③ *Build aborts immediately on error*—in contrast to some other case study build systems, problems encountered in building **gnat-gps** cause the build to abort immediately. The **gnatmake** tool is written in Ada, which supports exceptions and is known for its use in safety-critical software. The commands used in **gnat-gps** Makefiles are generally of limited complexity, rather than complex inline shell scripts.

Conclusion

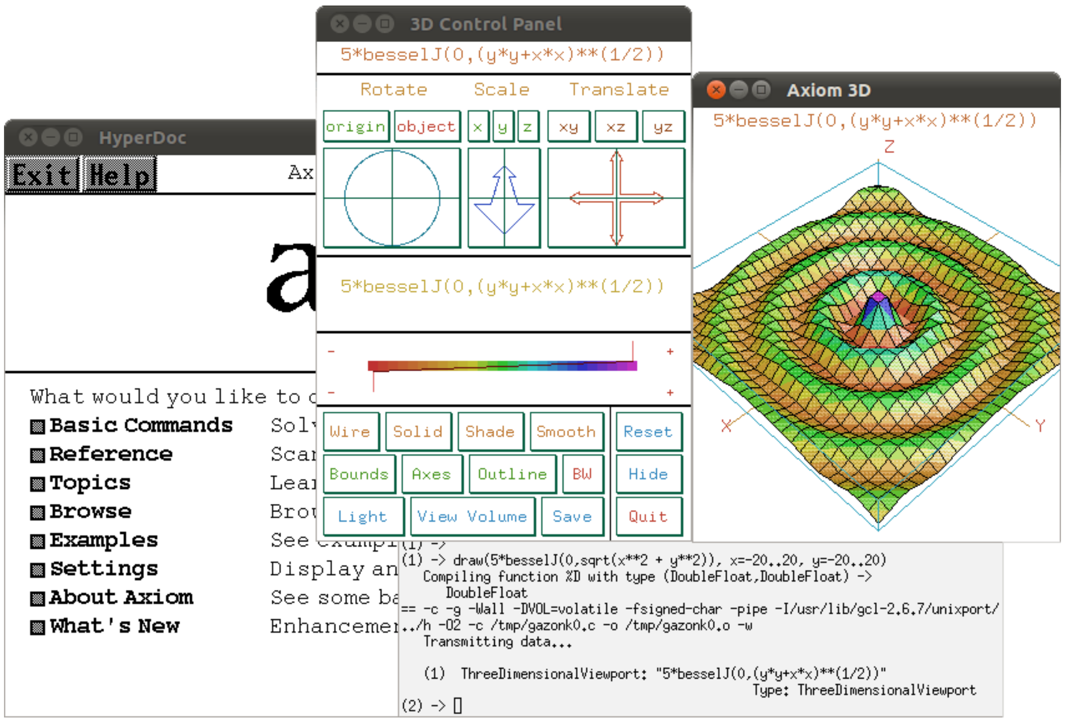
gnat-gps is difficult enough to build with the vendor build system that the Ubuntu maintainer wrote his own build system. However, this build system also has its own issues, which are somewhat understandable given that it was designed to designed to build system-installable binary packages for Ubuntu using a hardcoded configuration, rather than to be a portable general-purpose build system.

4.4 Case study: axiom

A summary of the findings for **axiom** is given in Table 4.4. The multilanguage diagram is Figure 4.16, and the build-time view diagram is Figure 4.20.

Question	Findings
Purpose and abstractions	Interactive computer algebra package with extensive library
Architecture, languages, and interactions	Compiler-interpreter for Scratchpad, written in Lisp, runs Scratchpad math library; C for portions of user-interface
Build system	Literate source built with literate Makefiles
Build issues	⚠1 Relies on broken GNU Common Lisp ⚠2 Does not build on Ubuntu 9.10 ⚠3 Build system inadvertently ignores some build errors ⚠4 No incremental rebuilds ⚠5 Build steps difficult to run manually
Rebuild issues	⚠4 No incremental rebuilds
Build features	① Interactive interpreter means most users and many developers do not need to use the build system
Conclusion	Does not build, and literate code of limited usefulness in debugging build problems

Table 4.4: Summary of findings for axiom



```
(1) -> draw(5*besselJ(0,sqrt(x**2 + y**2)), x=-20..20, y=-20..20)
   Compiling function %D with type (DoubleFloat,DoubleFloat) ->
      DoubleFloat
== -c -g -Wall -DVOL=volatile -fsigned-char -pipe -I/usr/lib/gcl-2.6.7/unixport/
../h -O2 -c /tmp/gazonk0.c -o /tmp/gazonk0.o -w
   Transmitting data...

   (1)  ThreeDimensionalViewport: "5*besselJ(0,(y*y+x*x)**(1/2))"
                                                 Type: ThreeDimensionalViewport
(2) -> ▯
```

Figure 4.15: `axiom` screenshot

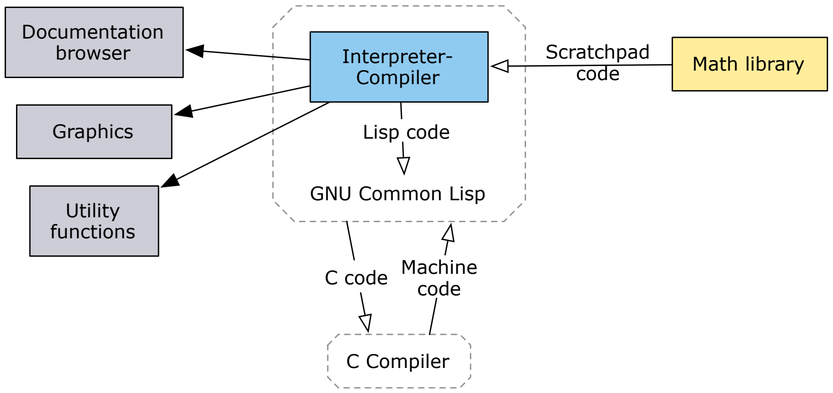


Figure 4.16: `axiom` multilanguage diagram, using the notation of Figure 4.1. This simplified diagram shows only a high-level view of the major parts of `axiom`. The Scratchpad interpreter-compiler is written in Lisp, and runs inside GNU Common Lisp (GCL). It compiles Scratchpad code to Lisp code. GCL in turn can either interpret the Lisp code directly, or compile it to C and call an external C compiler to generate object code that is then dynamically loaded into the running GCL/`axiom` process. The math routines of `axiom` are implemented in the Scratchpad language, and run inside the `axiom` environment. The internal data structure layouts of the interpreter-compiler are based on the structure of the mathematical categories defined in the math library. There is also a documentation browser, a graphics viewer, and a collection of utility functions written in C.

What does the package do? What are the major abstractions?

axiom is a computer algebra package that provides both an interactive environment for its own algebraic programming language called Scratchpad, and an extensive executable mathematical library. As shown in Figure 4.15, the main interface is a command prompt, but there is also a hypertext documentation browser and an interactive graphics viewer. A further example of the Scratchpad language is given in Figure 4.17. Scratchpad was originally developed in the 1970s by IBM for its mainframes [30], and was open-sourced in 2001 after being discontinued as a commercial project.

The math library is composed of three kinds of entities: packages, domains, and categories. Packages are collections of functions. For example, there is a package for computing the Laplace transform of a function, one for triangulating matrices, one for computing power series in function spaces, and more there are more than 800 other packages. Domains are packages that store data, and implement the mathematical structures that packages operate on. There are more than 400 categories implementing structures such as integers, polynomials, p-adic integers, Clifford algebras, and also computing science structures such as hash tables. Finally, categories are collections of axioms that define the type of a domain. For example, the category Abelian Semi Group extends the category Set with the axiom that there is an Abelian + operator.

Importantly, whether or not domains satisfy the axioms of the categories they declare themselves as belonging to cannot be and is not verified automatically. If a domain declares that its + operator is associative, that fact will be used by the compiler for optimization regardless of whether the operator actually is associative. To prevent incorrect results due to faulty assumptions, there are extensive regression tests to ensure that domains and packages behave as they are intended to, and that changes to other definitions do not cause incorrect results.

What are the major parts of the software, what languages are they written in, and how do they interact?

The major parts of axiom are the interpreter-compiler, the math library, and the GUI. A diagram showing the relationships between these parts, but not their internal structure, is given in Figure 4.16. The math library is written in Scratchpad. The interpreter-compiler for the Scratchpad language is written in Lisp. The graphics and documentation browser subsystems, as well as some utility functions, are written in C.

axiom compiles Scratchpad code into Lisp code. An example Scratchpad package, AddTwoNumbers, is shown in Figure 4.17. It defines a single function, addTwo, which adds two numbers. The compiled Lisp version is shown in Figure 4.18. It is essentially unrecognizable. The compiled Lisp code contains definitions of the axiom data structures pertaining

```
)abbrev package ADDTWO AddTwoNumbers
  AddTwoNumbers(A) : Exports == Implementation where
      A : IntegerNumberSystem
      Exports == with
          addTwo: (A, A) -> A
      Implementation == add
          addTwo(a, b) == a + b
```

Figure 4.17: Example Scratchpad package

```
(DEFUN |ADDTWO;addTwo;3A;1| (|a| |b| $)
  (SPADCALL |a| |b| (QREFELT $ 7)))

(DEFUN |AddTwoNumbers| (#0=#:G1763)
  (PROG NIL
        (RETURN
         (PROG (#1=#:G1764)
               (RETURN
                (COND
                 ((LETT #1#
                        (|lassocShiftWithFunction|
                         (LIST (|devaluate| #0#))
                         (HGET |$ConstructorCache| (QUOTE |AddTwoNumbers|))
                         (QUOTE |domainEqualList|)) |AddTwoNumbers|)
                  (|CDRwithIncrement| #1#))
                 ((QUOTE T)
                  (UNWIND-PROTECT
                   (PROG1 (|AddTwoNumbers;| #0#)
                          (LETT #1# T |AddTwoNumbers|))
                   (COND ((NOT #1#)
                          (HREM |$ConstructorCache|
                                (QUOTE |AddTwoNumbers|)))))))))))

(DEFUN |AddTwoNumbers;| (|#1|)
  (PROG (DV$1 |dv$| $ |pv$|)
        (RETURN (PROGN
                 (LETT DV$1 (|devaluate| |#1|) . #0=(|AddTwoNumbers|))
                 (LETT |dv$| (LIST (QUOTE |AddTwoNumbers|) DV$1) . #0#)
                 (LETT $ (GETREFV 9) . #0#)
                 (QSETREFV $ 0 |dv$|)
                 (QSETREFV $ 3 (LETT |pv$| (|buildPredVector| 0 0 NIL) . #0#))
                 (|haddProp| |$ConstructorCache|
                  (QUOTE |AddTwoNumbers|)
                  (LIST DV$1)
                  (CONS 1 $))
                 (|stuffDomainSlots| $)
                 (QSETREFV $ 6 |#1|) $))))


(MAKEPROP
 (QUOTE |AddTwoNumbers|)
 (QUOTE |infovec|)
 (LIST (QUOTE #(NIL NIL NIL NIL NIL NIL (|local| |#1|) (0 . +) |ADDTWO;addTwo;3A;1|))
       (QUOTE #(|addTwo| 6))
       (QUOTE NIL)
       (CONS (|makeByteWordVec2| 1 (QUOTE NIL))
             (CONS (QUOTE #())
                   (CONS (QUOTE #())
                         (|makeByteWordVec2| 8
                          (QUOTE (2 6 0 0 0 7 2 0 6 6 6 8))))))
       (QUOTE |lookupComplete|)))
```

Figure 4.18: `axiom` compiler output for Scratchpad code in Figure 4.17

```
static object LI1(V4,V5,V6)

object V4;object V5;object V6;
{         object V9;
        goto TTL;
TTL:;
        {object V7;
        V7= ((V6))->v.v_self[(long)7];
        {object V8 = (
        V9 = CMPcar((V7)),
        (type_of(V9) == t_sfun ?(*((V9)->sfn.sfn_self)):
        (fcall.argd=3,type_of(V9)==t_vfun) ?
        (*((V9)->sfn.sfn_self)):
        (fcall.fun=(V9),fcalln))((V4),(V5),CMPcdr((V7))));
        VMR1(V8)}}
        return Cnil;
}
```

Figure 4.19: Lisp compiler output for the `ADDTWO;addTwo;3A;1` function in Figure 4.18

to the package, and calls many internal `axiom` functions. The compiled `addTwo` function is defined on lines 1–2. It takes three arguments: the original arguments `a` and `b`, and a context argument `$`. The addition operation is specified by `QREFELT $ 7`; it corresponds to the + function highlighted on line 43 at zero-based index 7 in its containing list. This Lisp code bears a similar relationship to the original Scratchpad source code as compiled machine code does to corresponding source code in other high-level languages.

`axiom` uses GNU Common Lisp, or GCL for short, to run itself and to produce machine code. GCL was once known as Austin-Kyoto Common Lisp, or AKCL, and was derived from Kyoto Common Lisp. Some of its development was under contract from IBM specifically for Scratchpad. The Scratchpad interpreter compiler is written in Lisp, and compiles Scratchpad code to Lisp. While lisp code can be directly interpreted by GCL, for performance reasons it is preferable to compile it to machine code. GCL compiles Lisp to C code. The compiled version of the `addTwo` function from Figure 4.18 is shown in Figure 4.19—though nearly unrecognizable, it is still looking up the + operator as the seventh element of a package property and then calling it. Note that the function now has the generic name `LI1`, and is declared `static`—it will not be exported in the symbol table of the compiled code. Despite this, GCL compiles this to an unrelocated intermediate object file and is able to dynamically load it into the running executable, by implementing its own linking and loading routines. GCL records the fact that the non-exported function `LI1` is really `addTwo` by concatenating its own custom symbol table data structure to the end of the compiled object file, outside any section structure as defined by the Executable and Linking Format (ELF) file specification [13]. This GCL-specific data is even invisible to binary-file inspection tools like `objdump` which expect binary files to have custom data in custom file sections [40].

What is the structure of the build system and how is it implemented?

A build system diagram for `axiom` is given in Figure 4.20, and the build system structure is described in the caption of that figure.

Some background information about Lisp concepts and terminology may be helpful in understanding Figure 4.20. A Lisp environment, sometimes referred to as simply a Lisp, is an interface that, in a loop, interactively reads Lisp code, evaluates it, and prints the result. Lisp code evaluated by a Lisp environment can dynamically modify the Lisp environment itself by defining new symbols and redefining existing ones inside the Lisp environment. Large Lisp software projects can be developed interactively without complete textual source code for the software ever existing; the functions and data structures in the memory of the Lisp environment constitute the software. To preserve such software, Lisp environments provide a means to save all the definitions of a Lisp environment as a *Lisp image*. Usually, a Lisp image is an executable binary that starts up a Lisp environment prompt when run, but has all the definitions of previous Lisp session embedded inside the image. These can be distributed to other people, can be dynamically modified, and new derivative Lisp images can be produced from them.

`axiom` is implemented using the noweb literate programming system. There are over 1.4 million lines of LaTeX macro code, such as that in Figure 4.21. The source files are individually as large as 7 MB. These files can be compiled into more than 20 000 typeset pages of combined source code and documentation, such as the one in Figure 4.22, or into approximately 360 KSLOC of Lisp, C, and Scratchpad source code.

The maintainer chose to use a literate programming system because, when `axiom` was open-sourced, there were few source-code comments, and he wanted to preserve knowledge of how and why the package works [14].

Most of the build system is implemented as Makefiles written in noweb. Configuration is performed by selecting one of 37 different operating-system-specific top-level Makefiles generated by the noweb macro system. These top-level Makefiles contain hard-coded configuration parameters for each operating system.

What, if any, issues are there in building the package?

Many problems were encountered when attempting to build `axiom`, and in fact it does not build at all. The key reason is that `axiom` relies on GCL, which is not compatible with this version of Ubuntu. However, this key build issue is exacerbated by the build system inadvertently ignoring errors; the absence of incremental rebuilds; and that individual build steps are difficult to execute manually.

⚠ `axiom` *relies on a broken GNU Common Lisp*—The relationship between `axiom` and

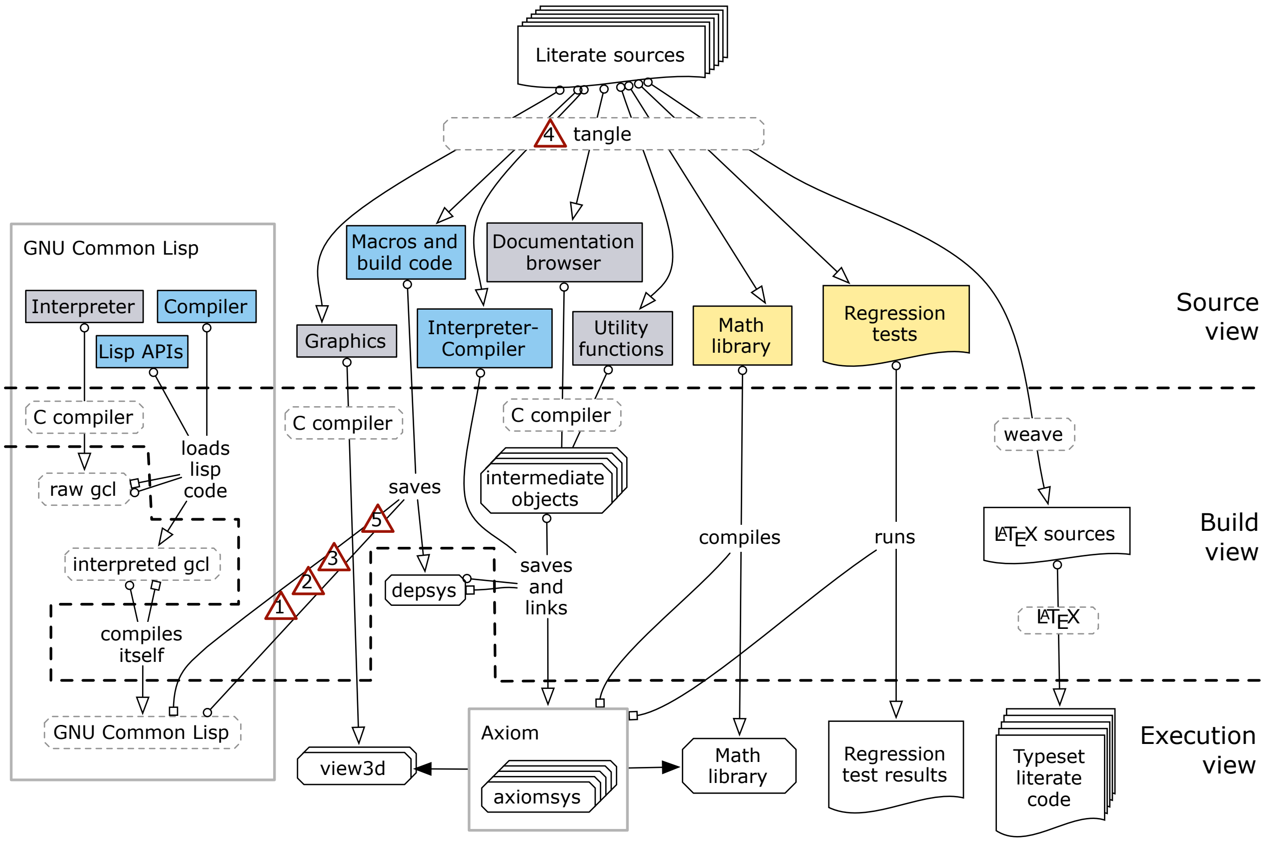


Figure 4.20: `axiom` build diagram, using the notation of Figure 4.1. A *Lisp image* is defined on page 61.

`axiom` relies on GCL, a Lisp environment written in a combination of C and Lisp. First a C-language interpreter for a subset of Lisp is compiled. It loads the Lisp code for the compiler and the Lisp APIs, and saves the resulting image. This image is a full-fledged Lisp interpreter-compiler, but running as interpreted code. It recompiles itself, producing the final compiled Lisp executable.

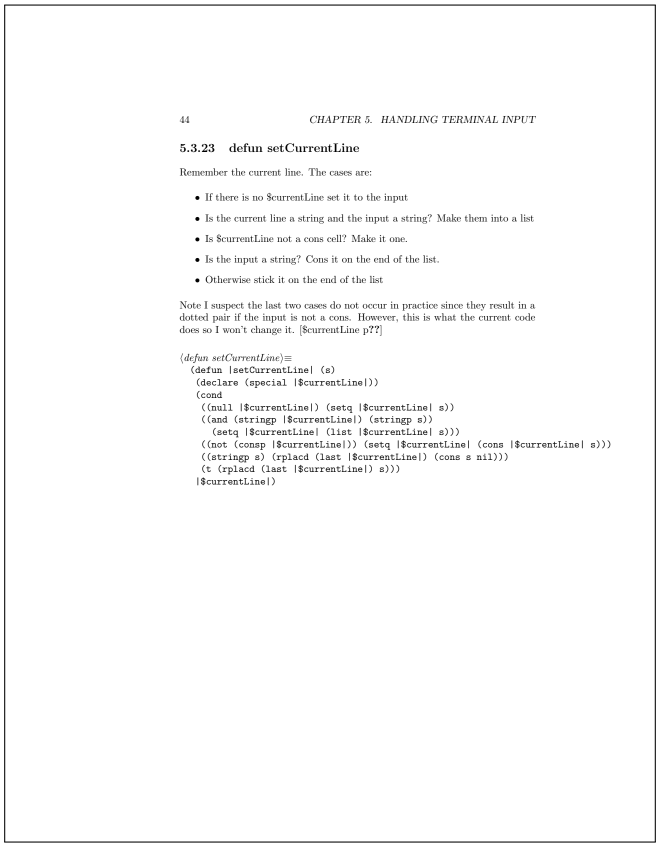
The source code for `axiom` is a collection of literate source files written in a macro-language dialect of LaTeX with fragments of source code in other languages. A tool called `weave` extracts the LaTeX markup and formats the other source code, producing LaTeX source files which are then typeset.

Source code in C, Lisp, and Scratchpad is also extracted from the source code. Some C code produces standalone binaries. Other source code is linked into Lisps containing `axiom` code. First, GCL saves a Lisp containing macro definitions and some build code for `axiom`. This Lisp compiles the interpreter-compiler for Scratchpad, producing the main `axiom` binaries. These in turn compile the math library written in Scratchpad, and then execute the regression tests, consisting of Scratchpad input and expected output.

At runtime, the `axiom` interface is implemented by eight instances of five different binaries, communicating via sockets.

```
\defun{setCurrentLine}{setCurrentLine}
Remember the current line. The cases are:
\begin{itemize}
\item If there is no \$currentLine set it to the input
\item Is the current line a string and the input a string?
      Make them into a list
\item Is \$currentLine not a cons cell? Make it one.
\item Is the input a string? Cons it on the end of the list.
\item Otherwise stick it on the end of the list
\end{itemize}
Note I suspect the last two cases do not occur in practice since
they result in a dotted pair if the input is not a cons. However,
this is what the current code does so I won't change it.
\usesdollar{setCurrentLine}{currentLine}
<<defun setCurrentLine>>=
(defun |setCurrentLine| (s)
 (declare (special |$currentLine|))
 (cond
  ((null |$currentLine|) (setq |$currentLine| s))
  ((and (stringp |$currentLine|) (stringp s))
    (setq |$currentLine| (list |$currentLine| s)))
  ((not (consp |$currentLine|)) (setq |$currentLine| (cons |$currentLine| s)))
  ((stringp s) (rplacd (last |$currentLine|) (cons s nil)))
  (t (rplacd (last |$currentLine|) s)))
 |$currentLine|)
```

Figure 4.21: noweb input example from `axiom` source code



44 *CHAPTER 5. HANDLING TERMINAL INPUT*

5.3.23 defun setCurrentLine

Remember the current line. The cases are:

- If there is no $currentLine set it to the input
- Is the current line a string and the input a string? Make them into a list
- Is $currentLine not a cons cell? Make it one.
- Is the input a string? Cons it on the end of the list.
- Otherwise stick it on the end of the list

Note I suspect the last two cases do not occur in practice since they result in a dotted pair if the input is not a cons. However, this is what the current code does so I won't change it. [$currentLine p**??**]

⟨*defun setCurrentLine*⟩≡
```
  (defun |setCurrentLine| (s)
   (declare (special |$currentLine|))
   (cond
    ((null |$currentLine|) (setq |$currentLine| s))
    ((and (stringp |$currentLine|) (stringp s))
      (setq |$currentLine| (list |$currentLine| s)))
    ((not (consp |$currentLine|)) (setq |$currentLine| (cons |$currentLine| s)))
    ((stringp s) (rplacd (last |$currentLine|) (cons s nil)))
    (t (rplacd (last |$currentLine|) s)))
   |$currentLine|)
```

Figure 4.22: Typeset noweb output example from `axiom` source code

GCL was explained above: the `axiom` interpreter-compiler is written in Lisp and compiles to Lisp, while GCL compiles Lisp to C, invokes a C compiler, and then executes its own linking and loading routines using its own symbol tables to dynamically load intermediate object code into the running Lisp process. The last release of GCL on its homepage[1] is from 2005, and there have been many changes to Linux since then. GCL appears to be currently unmaintained. There is a version control repository,[2] but there has been only one commit in the last two years, and that to a single file.

Ubuntu's precompiled GCL binary aborts with a segmentation fault on startup. It predates the now-common use of `exec_shield`, which makes writable memory regions non-executable to prevent malicious security exploits from loading executable code into memory, and Address Space Layout Randomization, which loads libraries at random locations in memory to prevent malicious security exploits from having well-known memory locations to target. GCL both attempts to execute code it has just written into memory, and persists library function addresses between executions. Disabling these security features allows GCL to run, but it is unable to compile `axiom`, reporting the error message `cannot read address Lisp initialization failed`.

`axiom` includes the source code for 3 different versions of GCL in its source distribution, and patches for 14 different versions in total. Attempts to use these fail again with the same `cannot read address` error message. On the `axiom` mailing list, the user Enrique Perez-Terron described both the cause and his solution [51], though he was unable to build the package even after resolving this issue. `axiom`'s build failure is a failure of GCL's `compiler::link` function, which is supposed to save a new Lisp image containing the current environment. Perez-Terron found that the failure occurred because GCL was unable to parse some newer statements in the memory map file output by the compiler's linker. One solution is to create a shim script with the same name as the linker that appears earlier in the executable search path. The shim script executes the real linker, then edits the resulting file to remove portions that GCL does not understand. This succeeds in overcoming the `cannot read address` error, only to produce the similar error message `I'm not an object Lisp initialization failed` instead.

GCL's inability to save compiled code is likely the result of the low-level custom operations it performs on object files. GCL implements its own nonstandard symbol table format, and its own linking and loading mechanisms. These have no longer compatible with updates to the standard linking and loading mechanisms of Linux, or the output of tools that support those mechanisms.

2 `axiom` *does not build at all on Ubuntu 9.10*—while the vendor build system was

[1]www.gnu.org/software/gcl/
[2]http://cvs.savannah.gnu.org/viewvc/?root=gcl

originally used, the Ubuntu-supplied patches and build scripts were examined after repeated failures of the vendor build system. Unlike the case with `gnat-gps`, there was no significant difference between the build systems, and the Ubuntu build failed to produce a working binary. One portion of the Ubuntu build script involves a mysterious attempt to start the `axiom` build; wait 30 minutes; then kill the build and restart it.

The incompatibility of GCL on Ubuntu 9.10 is irreconcilable at the build level, and makes `axiom` unbuildable. A significant amount of development work to update this version of GCL to work on Ubuntu 9.10 would be necessary to make it build.

An attempt to build the newer Ubuntu-supplied version of `axiom` on Ubuntu 10.10 did succeed on the first try, however; that is what was used to generate screenshots and make the investigations described earlier in this section.

While `axiom`'s unbuildability is due to an incompatibility between GCL and the system linker, the following build issues exacerbated determination of that cause.

The build system inadvertently ignores build errors—The Makefiles contain many complex multiline shell commands that lack proper error handling. When these fail, the build proceeds anyway, only to fail later when the earlier build output is found to be missing, resulting in error messages such as `Command not found`.

Consider the Makefile fragment in Figure 4.23.

```
1  gcldir:
2          # .. commands to patch GCL source code omitted ...
3          @(cd ${GCLVERSION} ;
4          ./configure ${GCLOPTS} ;
5          ${ENV} ${MAKE} ;
6          echo '(progn (load "cmpnew/gcl_collectfn.lsp")'
7           '(load "lsp/sys-proclaim.lisp") (compiler::emit-fn t)'
8           '(system::save-system "${OUT}/lisp"))' | unixport/saved_gcl )
9          @echo 13 finished system build on `date` | tee >gcldir
```

Figure 4.23: `axiom` makefile fragment

This Makefile rule has no prerequisites, and its effect is to enter the GCL source directory; configure it; build it; and then attempt to save a new Lisp image by running the newly-built GCL. The statements are joined by the shell's '`;`' sequential list operator, which executes all commands regardless of their error return value, and returns the error return value of only the *last* command executed to Make. If the directory does not exist, if the configure fails, of if the GCL build fails, then the resulting error message is `command not found` when `saved_gcl` cannot run on line 8. On the axiom mailing list, users asking for help with build problems have posted apparent error messages, only to receive replies stating, "Your build failure originated much earlier in the log – could you please forward the entire thing?" [41]

Even if the build of GCL succeeds, the `save-system` may not—as described above, compilation is broken for GCL running on this version of Ubuntu, and the `emit-fn` setting enables the compiler, causing the `save-system` to fail. However, GCL does not propagate the failure of the `save-system` call back to the shell as a return error value, and the build of `axiom` proceeds as if GCL was successfully built, only to fail much later with a different `command not found` error message when the Lisp built here is not found.

This problem occurs in other `axiom` Makefiles as well. While these Makefiles could be modified to use the conditional '`&&`' operator instead, there are other issues with this fragment.

⚠4 *There is no support for incremental rebuilds*—The majority of `axiom`'s source code is written in the noweb macro language, and expanded at build time to produce individual source files. Using `axiom`'s build system, this procedure can only be performed *en masse*, expanding every source file. Even if changes to the noweb source file result in changes only to the typeset documentation, or to only a few expanded source files changing, every possible source file is newly-expanded from the noweb source and receives an updated timestamp, causing Make to rebuild hundreds of source files instead of just a few changed ones.

Another example of lack of support for incremental builds is found in Figure 4.23, where the steps to build GCL are all chained together as one unit. If modifying GCL, for example to insert tracing code to debug the linking problems described above, then this Makefile fragment results in every compilation triggering a time-consuming full reconfiguration of GCL. Since the configuration process writes out a header file containing inferred configuration values which is `#include`d directly in every header file and indirectly in every source file, this also triggers a full rebuild of all of GCL.

The inability to make a small change and quickly see its effect, to say nothing of its effect on the package development process, makes it difficult to debug build problems.

⚠5 *Build steps are difficult to run manually, in part due to use of environment variables for configuration*—when debugging build problems, it is often necessary to manually try variations of failing build commands until they succeed. `axiom`'s build system makes such debugging difficult, in part due to coarse-grained build targets, and in part due to use of environment variables for configuration.

One example of such a coarse-grained target is the mass-extraction of noweb source code for a major source code component. Another example occurs in Figure 4.23. The target is a custom version of of GCL. Building the target requires configuring GCL, building GCL, and then loading custom code into it and saving a new Lisp image. All of these operations are treated as a single step, and the Makefile provides no way to run only one of them. Experimenting with variations of the final image-saving step cannot be done with this Makefile as-is.

The problem of coarse-grained targets is exacerbated by extensive use of environment variables for configuration. When individual build steps are simple commands such as running a script or tool with one or two arguments, the build system's inability to pinpoint build steps is not a significant barrier, as a developer can manually execute the build commands by typing them in. However, many of `axiom`'s build commands are more complicated scripts such as the Lisp script in lines 6-8 of Figure 4.23. This script references the environment variable `${OUT}`, one of 25 environment variables set by top-level Makefiles. These environment variables specify not only paths to tools and source code directories, but also Makefile control flow and build system logging. The variables are propagated through a hierarchy of 23 different Makefiles, and also used internally by some scripts called by the build system. Manually running a build command that depends on these environment variables requires all of them to be set correctly. While the environment variables are echoed to the terminal at some points of the build, they contain spaces that are not properly escaped, so copying-and-pasting them directly into the shell is not possible.

Build systems such as `python3.0`'s and `synopsis`'s use autoconf to store this configuration data in files instead of environment variables. When there are hierarchies of Makefiles, the configuration parameters are textually substituted into each Makefile, but can be overridden from the Make command line when necessary.

What, if any, issues are there in building changes to the package?

As discussed above as build issue ⚠4 , `axiom`'s build system does not support incremental rebuilds of itself.

However, there is full and excellent support for interactively rebuilding mathematical code written in Scratchpad, so the lack of incremental rebuilds for the `axiom` environment is not an issue for most end-users, and for many developers working on the mathematical side of `axiom`. The mathematical library is implemented entirely in Scratchpad, and, once an `axiom` binary is available, users and developers are free to interactively develop new mathematical domains and mathematical packages.

What build system features prevent build problems similar to those encountered in other packages?

① *Because* `axiom` *is an interactive programming environment, the build system and its problems do not affect most users and many developers*—the interactive nature of the `axiom` environment means that end-users and developers wishing to do mathematics or implement mathematical algorithm with `axiom` do not use the build system to do so. As long as some pre-compiled binaries are available somewhere, whether from an operating system distribution repository or from the `axiom` web site, `axiom` can be used and extended. While some users will want to build the software themselves, only those wishing to modify the internals of the Scratchpad interpreter-compiler need to deal with the build problems.

Conclusion

`axiom`'s high-level architecture shown in Figure 4.16 makes `axiom` seem simple. But the build system is extremely complicated, with nonexistent tool support, and no use of frameworks or abstraction in the build system implementation. It can not be built on Ubuntu 9.10, due to its reliance on an incompatible version of the Lisp implementation GCL, and this key problem is exacerbated by many other build system issues.

The use of literate programming for the implementation was of limited help in debugging the numerous `axiom` build issues. It is voluminous, verbose, and outdated. For example, `src/interp/Makefile.pamphlet` contains several definitions of variables whose names have an "O" prefix; there is a comment with each that "O" stands for "old," but, to quote the file, "The use of 'old' is something of a sublet concept as there were several generations of 'old' and all meaning of the term is lost." The corresponding variables are all set to blank values and unused. There are extensive references to a Lisp dialect called Boot that parts of `axiom` were once implemented in, which required additional build support. Despite the fact that it is apparently not used anymore, there are lengthy discussions of it throughout the build system source.

There are potentially valid reasons for these properties of the literate source code. `axiom` was not developed as a literate program, but rather, open-sourced with minimal comments. As described in Section 0.3.2 of Volume 4 of the source, the current maintainer was unable to build the original open-source release at all, as it had been ported to the Lisp implemented Codemist Common Lisp. The maintainer ported it back to GCL, and chose a literate format in an attempt to preserve the limited remaining knowledge of how and why `axiom` works [14].

Overall, the dynamic interactivity of the `axiom` environment means that, once built, build and rebuild problems matter little to end-users or developers doing mathematics in and extending the capabilities of `axiom`, because doing so takes place entirely within the `axiom` environment.

4.5 Case study: `ruby-prof`

A summary of the findings for `ruby-prof` is given in Table 4.5. The multilanguage diagram is Figure 4.24 and the build-time view diagram is Figure 4.25.

What does the package do? What are the major abstractions?

`ruby-prof` is a profiler for the Ruby programming language. It can measure and report how much time, memory, and/or 'garbage' is spent in different parts of Ruby-language soft-

Question	Findings
Purpose and abstractions	Profiler for Ruby source code
Architecture, languages, and interactions	Ruby script and Rake build plugin using Ruby library, with C extension for collecting profile data
Build system	Rake-based with auto-generated Makefile for C extension
Build issues	⚠1 Build happens at install time ⚠2 Unavailable options are exposed, causing crashes ⚠3 No mechanism to keep windows binary up-to-date ⚠4 Only works with compiler used to build Ruby
Rebuild issues	⚠5 No dependencies in auto-generated build system
Build features	① Exports reusable general-purpose build functionality
Conclusion	Small packages have build issues too. Object-oriented build systems show promise but may lack maturity

Table 4.5: Summary of findings for `ruby-prof`

ware. ‘Garbage’ consists of no-longer-referenced objects that are reclaimed by the garbage collector.

What are the major parts of the software, what languages are they written in, and how do they interact?

As shown in Figure 4.24, there are four parts to `ruby-prof`. The `RubyProf` library handles analysis and formatting of profiling data collected by a profile data collection module. The functionality can also be accessed via a command-line user interface, or via a build plugin that can integrate profiling into the builds of other Ruby-language packages.

`ruby-prof` is written in Ruby, except for the profile data collection, which is written in C. The nature of the interaction is similar to that between Python modules implemented in C and Python, as described starting on page 39. The Ruby interpreter is written in C and provides C-language APIs equivalent to those available to Ruby-language code. Although implemented in C, `ruby-prof`’s profile data collection module calls methods on Ruby objects, and implements methods of Ruby object classes. The implementation language of a Ruby class is an implementation detail hidden from clients.

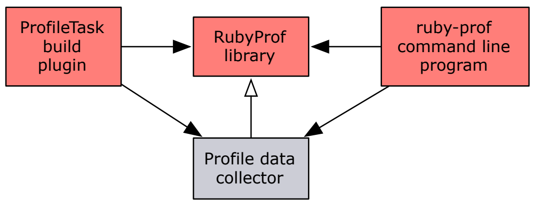


Figure 4.24: `ruby-prof` multilanguage diagram, using the notation of Figure 4.1. `ruby-prof` can be used as a command-line tool, a build-time plugin, or as a library. The build-time plugin and command-line user interface call the library to analyze and format profile data. For speed reasons, the collection of profile data is implemented in C.

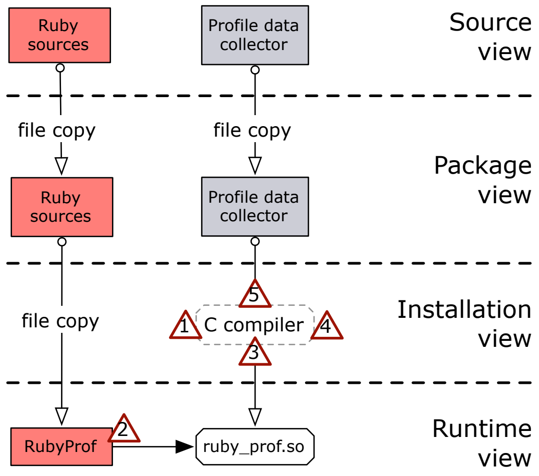


Figure 4.25: `ruby-prof` build diagram, using the notation of Figure 4.1. Parts implemented in Ruby are copied unchanged into the execution environment. The profile data collection module, consisting of a single C source file, is compiled into a shared library installation time.

`ruby-prof` is a small package, totalling only 4 KSLOC, including 1 KSLOC of unit tests. The profile data collection module consists of a single C implementation file and several C header files.

What is the structure of the build system and how is it implemented?

A diagram of `ruby-prof`'s build structure is given in Figure 4.25. The Ruby-language parts are copied to the execution environment, and the C part is a single implementation source file that is compiled against the Ruby headers and library to produce a shared library loaded at runtime.

Before describing the implementation of `ruby-prof`'s build system, it is first necessary to provide some background information about the Rake build tool that it uses.

Background: Rake

Rake is an object-oriented replacement for Make written in the Ruby programming language. Fowler provides an enthusiastic overview [20]. It differs from Make in that its Rakefiles are Ruby-language source code, with full access to Rake's object model. This allows dynamic customization and extension of the build system. While many other build tools are either task-based or artifact-based but not both, Rake supports any paradigm that can be implemented in Ruby by subclassing the Rake library's `Task` object. This includes artifact- and task-based builds. For example, the basic artifact-based Makefile

```
.PHONY: default

default: foo
```

```

foo: foo.c
        gcc $^ -o $@
```

has a straightforward transliteration as the Rakefile

```
task :default => 'foo'
  file 'foo' => 'foo.c' do |t|
    system *["gcc", *t.prerequisites, "-o", t.name]
end
```

which is Ruby source code. The `File` task implicitly uses file timestamp dependency consistency checks. However, instead of being forced to use Make's paradigm of file timestamp checks for consistency between individual files, the Rakefile-based build system can be customized by subclassing `Task` or importing subclasses from other libraries. Tasks that represent parts of files, aggregations of files, tasks, or other abstract entities can be used, with dependency consistency checked by any desired computation. For example, `ruby-prof` provides a `ProfileTask` subclass to automatically profile unit tests at build time.

`ruby-prof`'s build system implementation

The default build target produces an installable Ruby `.gem` package file that is installed by the standard Ruby installation tool also called `gem`.

Surprisingly, `ruby-prof`'s profile data collection module, which is implemented in C, is not built during the build, and there is no Rakefile build target to build it. Instead, during the build, the need for the module to be built is recorded in metadata. The `gem` tool reads this metadata at install time, and executes a non-Rake `ruby-prof` build script specifically for building the profile data collection module. This non-Rake build script uses a different Ruby build tool, namely the make-Makefiles tool called `mkmf`, to auto-generate a Make-based build system. The auto-generated Make-based build system is then executed by `gem`.

Developers wishing to manually build the C parts of `ruby-prof` can run the build script specified in the package metadata to auto-generate the Make-based build-system for it, and then use Make.

What, if any, issues are there in building the package?

Getting `ruby-prof` to build and run initially was problematic, as the build system unexpectedly defers compilation until install time. Some optional features disabled at configuration time are still exposed through the UI, but attempting to select them crashes the software. Additionally, there are issues with possibly inconsistent binaries distributed with the source code, and limited flexibility of the build system.

⚠ *The build happens at install time*—when a developer downloads the source code for `ruby-prof` and runs the build system, the necessary shared library that is part of it is not actually built. If the software is not installed, then trying to run it produces the error message "`no such file to load -- ruby_prof.so`." If `ruby-prof` is already installed, then the shared library from the installed version is used, even if it is incompatible. The software may crash, and if it does not, any recent changes to the `ruby-prof` source code will not be in effect.

The `gem` tool that compiles the shared library at installation time does not provide an option to compile a development version. Developers can manually run a script to generate a Makefile, and then run Make.

△2 *Unavailable options are exposed, causing crashes*—some `ruby-prof` features, such as profiling garbage collection, require a patched Ruby interpreter that includes extra instrumentation hooks. When building the profile data collection module at install time, the needed hooks are checked for by the auto-generated build system. The `mkmf` tool checks for the availability of these hooks by test-compiling a sample program, and defines appropriate preprocessor symbols for each one if found. This is the same strategy used by autotools. The profile data collection module then uses the preprocessor symbols to conditionally compile support for only the hooks available in the Ruby runtime.

However, the `RubyProf` library and the `ruby-prof` command-line tool do not take this configuration information into account. The help for the command-line tool indicates parenthetically that using certain profiling modes "`(requires a patched Ruby interpreter)`." Trying to use unavailable modes causes `ruby-prof` to crash. Patching the Ruby interpreter and reinstalling it is also not enough; since the checks are performed at `ruby-prof` installation time, `ruby-prof` must be reinstalled too.

This issue could be resolved with changes to `ruby-prof`. The build system could be modified to propagate configuration information back to the Ruby parts of `ruby-prof`. Or, the conditionally-compiled code could be modified to abort with an appropriate error message instead of crashing when required instrumentation hooks in the Ruby runtime are missing.

△3 *There is no mechanism to keep the windows binary up-to-date when building on other platforms*—compilers are usually not present on the windows platform, causing complications with the compilation-at-installation issue described earlier. `ruby-prof` compensates by including a windows `.dll` binary shared library as part of the source code distribution. The `.dll` is a compiled version of the C-language profile data collection module, labelled `ruby_prof.so` in Figure 4.25. This binary is automatically used, even if a compiler is available at install time, unless the `--platform=ruby` option is specifically passed to the `gem` tool at install time.

However, there is no mechanism for keeping this Windows binary up-to-date with the source code. Developers modifying and releasing new versions of `ruby-prof` must remember to manually find a Windows machine or cross-compiler to build an updated Windows binary.

This particular issue appears to be partially resolved in newer versions of `ruby-prof` than that examined for this case study. In the latest versions of `ruby-prof`, there is no longer a precompiled Windows shared library included with the source code. While some official source releases of `ruby-prof` are accompanied by Windows binary releases[1] and some are not, the situation is improved in part due to evolution of the Ruby ecosystem. Installations of versions of `ruby-prof` for which no Windows binary version has been uploaded now give a useful error message with URLs of download packages and installation instructions for the Ruby DevKit, a package containing header files and a self-contained compiler for building C parts of Ruby software.

⚠4 *The build system only supports the exact compiler used to build the Ruby interpreter*—the `mkmf` build system generator queries the standard Ruby `rbconfig` library for configuration information such as compiler options. However, the configuration information in `rbconfig` is fixed at the time that Ruby is compiled. It may not be appropriate for builds of C modules to be loaded into Ruby, and this limits flexibility.

In particular, on Windows, where the standard Ruby interpreter packages are built with the Microsoft C compiler, the GCC compiler is much more common for building C modules for Ruby, as it is freely available. The two compilers produce mutually compatible output files, but have very different command-line syntaxes. The `rbconfig` library provides only the Microsoft command-line options. To enable use of GCC on Windows, `ruby-prof` has a separate build script for Windows with its own hard-coded compiler options. This is in contrast to older build tools such as autotools which dynamically probe for compatible compilers and command-line options.

What, if any, issues are there in building changes to the package?

⚠5 *There are no header file dependencies in the auto-generated build system for the C part of* `ruby-prof`—the `mkmf` build tool auto-generates a Make-based build system for the C part of `ruby-prof`. The C part consists of one `.c` implementation file and nine `.h` header files. The header files in question have implementation code for various functions, not just headers.

The auto-generated Makefile has no dependencies on any of the header files. Developers who change the implementation code in the header files and then rerun Make will not have their changes incrementally compiled. Instead, Make will say, "`Nothing to be done`."

This is a bug in `mkmf`, which should automatically generate dependencies, or support adding them to its generated Makefiles. This issue may be partly because this particular

[1]The versions mentioning 'mswin' or 'mingw' at http://rubygems.org/gems/ruby-prof/versions

build system seems to be designed to only be run at install time. For single-use build systems that do not need incremental recompilation, such dependencies are not necessary; but then it seems strange to base the build system on Make, whose sole purpose is to enable incremental recompilation.

What build system features prevent build problems similar to those encountered in other packages?

① `ruby-prof` *exports general purpose build functionality in the form of its* `ProfileTask` *build plugin*—while other build systems examined in these case studies exhibit varying levels of modularity, none exhibit modularity to the extent of `ruby-prof`'s Rake-based system. It allows packages to export build-level functionality in a modular way to other packages. Specifically, the `ProfileTask` class that is part of `ruby-prof` extends Rake's default unit-testing task by adding the ability to profile unit test runs. Clients can add this feature to their own builds without extensive reworking of build systems, by adding an import for `ProfileTask` and use it in place of or in addition to Rake's default `TestTask`. Normally, a client package using `ProfileTask` would declare its dependency on `ruby-prof` in its metadata, and the `gem` tool would use this metadata to automatically install `ruby-prof` when it installs the client package.

Use of such modular extensions to build systems may be able to reduce the complexity encountered in other case studies.

Conclusion

Although `ruby-prof` is quite small when measured by lines of source code, it still has numerous build system issues. The lack of rebuilds when `.h` header files change is a serious issue. The compiler flexibility and windows binary loss of synchronization issues are potential issues noticed during investigation of the build system, rather than problems which broke the build. The build-at-install feature is unusual, but once it is understood, it is not a showstopper.

The extensibility features offered by Rake appear promising. However there appears to be a lack of maturity in Ruby build tools, exemplified by that absence of dependencies on header files. It seems likely that the situation will improve with time as the Ruby ecosystem matures; some of the issues with `ruby-prof`'s Windows binaries have already been addressed by library upgrades.

Chapter 5

Build patterns and anti-patterns

Comparative analysis of the build problems experienced in some case studies but not experienced in others yields knowledge about how the packages not experiencing problems avoided them. These findings are summarized in terms of build *patterns* [23] and *anti-patterns* [12]. A summary of which patterns and anti-patterns are based on which build problems and features identified in the case studies is given in Table 5.1; a more detailed version is given in Table 5.2; and the reverse table, showing which build issues and features give rise to which patterns and anti-patterns, is given in Table 5.3.

The following template is used to structure the findings for each pattern and anti-pattern:

- *Description*: What is the pattern or anti-pattern?
- *Consequences*: What results can come from following or avoiding the pattern or anti-pattern?
- *Evidence*: What evidence for the pattern or anti-pattern is there in the case studies?
- *Remedies*: What fixes can allow individual builds to succeed despite the presence of an anti-pattern? What are some steps that can be taken to incorporate a pattern into existing software?

Pattern or Anti-pattern	`synopsis`	`python3.0`	`gnat-gps`	`axiom`	`ruby-prof`
Anti-pattern: Filename Collision	△2				
Anti-pattern: Installation Required	△1	①	△6		△1
Anti-pattern: Unverified Third-Party Software	△3		△3 △4	△1 △2	
Anti-pattern: Incorrect Dependencies	① ②	△1	②		△3 △5
Anti-pattern: Ignored Error	④		③	△3	
Pattern: Build-Free Extensibility		②	①	①	
Pattern: Object-Oriented Builds	③	④			① △1 △5
Pattern: Persistent Configuration		③	△2 △7	△5	△2 △4

Table 5.1: Abbreviated build and rebuild issues and features

Pattern or Anti-pattern	`synopsis`	`python3.0`	`gnat-gps`	`axiom`	`ruby-prof`
Anti-pattern: Filename Collision	⚠2 Documentation build fails on case-insensitive filesystems				
Anti-pattern: Installation Required	⚠1 Running built software requires manual path setup	① Checks if running from build directory	⚠6 Blank icons, dialogs when run from build directory		⚠1 Build happens at install time
Anti-pattern: Unverified Third-Party Software	⚠3 PDF documentation does not build due to problem in third-party tool		⚠3 Incompatible project file syntax ⚠4 Inconsistencies between Ada and system interfaces	⚠1 Relies on broken GNU Common Lisp ⚠2 Does not build on Ubuntu 9.10	
Anti-pattern: Incorrect Dependencies	① Uses automatic dependencies ② Uses shared libraries	⚠ Manual dependency management result is that changing some header files forces a full rebuild, while changes to other header files are ignored	② `gnatmake` handles dependencies automatically		⚠3 No mechanism to keep windows binary up-to-date ⚠5 No dependencies in auto-generated build system
Anti-pattern: Ignored Error	④ Build aborts immediately on error		③ Build aborts immediately on error	⚠3 Build system inadvertently ignores some build errors	
Pattern: Build-Free Extensibility		② Can extend with Python and C plugins	① Scripting interface allows build-free extension	① Interactive interpreter means most users and many developers do not need to use the build system	
Pattern: Object-Oriented Builds	③ `distutils`-based build with autotools subbuilds	④ Standard `distutils` tool used to build standard library			① Exports reusable general-purpose build functionality ⚠1 Build happens at install time ⚠5 No dependencies in auto-generated build system
Pattern: Persistent Configuration		③ Uses autoconf for configuration	⚠2 Omits search paths for many third-party C libraries ⚠7 Python console does not show output	⚠5 Build steps difficult to run manually	⚠2 Unavailable options are exposed, causing crashes ⚠4 Only works with compiler used to build Ruby

Table 5.2: Build and rebuild issues and features

Issue or feature		(Anti-)pattern
`synopsis`		
⚠1	Running built software requires manual path setup	IR
⚠2	Documentation build fails on case-insensitive filesystems	FC
⚠3	PDF documentation does not build due to problem in third-party tool	UTPS
①	Uses automatic dependencies	ID
②	Uses shared libraries	ID
③	`distutils`-based build with autotools subbuilds	OOB
④	Build aborts immediately on error	IE
`python3.0`		
⚠1	Manual dependency management result is that changing some header files forces a full rebuild, while changes to other header files are ignored	ID
①	Checks if running from build directory	IR
②	Can extend with Python and C plugins	BFE
③	Uses autoconf for configuration	PC
④	Standard `distutils` tool used to build standard library	OOB
`gnat-gps`		
⚠1	References Ada libraries no longer part of the project	
⚠2	Omits search paths for many third-party C libraries	PC
⚠3	Incompatible project file syntax	UTPS
⚠4	Inconsistencies between Ada and system interfaces	UTPS
⚠5	Ubuntu maintainer rewrites build system	
⚠6	Blank icons, dialogs when run from build directory	IR
⚠7	Python console does not show output	PC
①	Scripting interface allows build-free extension	BFE
②	`gnatmake` handles dependencies automatically	ID
③	Build aborts immediately on error	IE
`axiom`		
⚠1	Relies on broken GNU Common Lisp	UTPS
⚠2	Does not build on Ubuntu 9.10	UTPS
⚠3	Build system inadvertently ignores some build errors	IE
⚠4	No incremental rebuilds	
⚠5	Build steps difficult to run manually	PC
①	Interactive interpreter means most users and many developers do not need to use the build system	BFE
`ruby-prof`		
⚠1	Build happens at install time	IR, OOB
⚠2	Unavailable options are exposed, causing crashes	PC
⚠3	No mechanism to keep windows binary up-to-date	ID
⚠4	Only works with compiler used to build Ruby	PC
⚠5	No dependencies in auto-generated build system	ID, OOB
①	Exports reusable general-purpose build functionality	OOB

Table 5.3: Use of build and rebuild issues in (anti-)patterns

- *Applicability*: Is this (anti-)pattern a general software implementation issue, a general build system issue, or an issue specific to build systems for multilanguage software?

  This template heading helps to identify build patterns and anti-patterns that could apply to build systems for single-language software as well.

In general, the build problems exemplify an anti-pattern and the build features help to resolve it. The *Evidence* template heading for each (anti-)pattern clarifies the relationship.

5.1 Anti-pattern: Filename Collision

Description

Different source files have similar names that can be confused on certain filesystems, causing build or run problems. For example, two files with identical names modulo case cannot both exist on case-insensitive filesystems; files with similar names modulo case can be mistaken for each other by build tools designed for case-sensitive filesystems; and different filesystems have different definitions of what characters are legal in filenames.

Consequences

Developers may be unable to extract or build source code, and error messages may obscure the cause of the problem.

For straightforward case collisions, such as files named `Readme` and `README` in the same directory, error messages may vary. A non-systematic examination of a few common tools shows that only one identifies the problem. The common `tar` unarchiving tool uses the last file present in the archive, silently overwriting any others. The `unzip` tool prompts for whether to replace an existing file. The Subversion version-control tool will display "`No such file or directory`" and "`working copy locked`" error messages. The Mercurial version-control tool, however, prints a meaningful error message: "`abort: case-folding collision between readme and README`."

There are other types of filename collision other than straightforward case collisions, such as the issue with the `Synopsis` directory and `synopsis.py` file described on page 36. That issue results in the error message, "`ImportError: cannot import name config`" which does not not identify the underlying problem. Other filename collision issues may involve similar error messages.

This is not strictly a Windows or Mac OS problem. The `synopsis` issue was encountered in Ubuntu. An operating system using a non-default filesystem is not a contrivance, due to extensive use of network filesystems, dual-booting, USB keys, and virtual machines.

Evidence

The `synopsis` documentation fails to build in Ubuntu when using a Mac OS filesystem over the network, as the case-insensitive filesystem causes the `synopsis.py` build script to be mistaken for the `Synopsis`. The Python code `from Synopsis import config` directs Python to load the `config` object from the `Synopsis` module, which could be a single file or a directory. Before checking for the intended `Synopsis` directory, Python first attempts to load a single-file `Synopsis.py` file; on the case-insensitive Mac OS filesystem, this succeeds, returning the `synopsis.py` build script instead. The build script does not contain a `config` object and the build aborts with an error.

To avoid this issue the Linux case-sensitive `ext4` filesystem was used to complete the `synopsis` build, and for the other case studies. However, an additional attempt was made to build the other case studies in Ubuntu, but using a Mac OS filesystem. While no further filename collisions were encountered, both the Brenta build system for `gnat-gps` and the `axiom` build system contain commands to hard-link files. Since hard-linking files is not supported in the specific configuration used to mount the Mac OS filesystem in Ubuntu, these commands failed with `Operation not permitted` error messages.

Remedies

Once the problem is known, the remedy is to rename the files so that they no longer collide.

Automated detection of known classes of conflicts, such as straightforward case collisions or Python module directories with the same name, modulo case, as single Python files, can be quickly implemented by writing a short script.

One method of detecting the problem that can be implemented today, and can detect all kinds of case-collision issues, is to regularly test-build the software on a variety of platforms and filesystems. Even when a project does not support a specific platform, it may be useful to perform basic tests, such as whether it is possible to unpack the source code on other platforms without error.

Applicability

This anti-pattern is specific to build systems, but is not specific to multilanguage software. It is possible that cross-language filename conflicts, such as differences of legal filenames between programming languages, could be an issue, but this was not experienced in any of the case studies.

5.2 Anti-pattern: Installation Required

Description

Software does not work as soon as executables are built, but requires an additional installation stage. The uninstalled executables might not run at all, might have broken features, or could produce error messages.

Consequences

End-users and new developers performing their first builds may believe they have done something wrong when their just-built software does not run. For all developers, it is time-consuming to re-install packages after every compilation, and frustrating for source changes not to take effect when developers forget to reinstall. Installation may require administrator privileges. Development versions of software can be unstable, untested, and/or incompatible, and installing such software can interfere with day-to-day use of a computer.

Evidence

Running a newly-built but uninstalled version of `synopsis` requires extensive manual path setup for the different parts of `synopsis` to be able to load each other.

When `gnat-gps` is not installed, a version run from the build directory displays blank dialogs in place of necessary functionality. Dialog template resources are spread throughout different source directories and aggregated at install time, but the resource-loading routines can only load from a single directory.

In contrast, `python3.0`'s source code checks to see if it is running from a build directory, and, if so, adjusts search paths accordingly so that the development version is fully functional yet independent from any other instances of Python on the system.

Running the top-level build command for `ruby-prof` does not build all parts of the software; the C part is compiled at installation time. Since the build system for the C part is designed to run only once at install time, it is missing key rebuild features such as dependency-checking for header files, as described under the *Incorrect dependencies* pattern.

Remedies

Using a source code or build output directory structure that mirrors the expected deployment directory structure is one option. This approach may not scale for larger projects where artifacts are divided into different directories for different logical components. In such cases, an approach similar to `python3.0`'s internal adjustments of its search paths when running from the build directory can be used.

The negative consequence of installed development versions of software interfering with day-to-day work can be mitigated somewhat by installing the software into a virtual machine, or a virtual-machine-like environment such as a Linux `chroot` or FreeBSD `jail`. However, in addition to requiring extensive additional setup procedures, requiring developers to have superuser privileges, being non-portable, and creating security risks, these mechanisms all still require time-consuming installation procedures in which software does not work as soon as executables are built.

Applicability

This is a general build system issue, but may be more pronounced in multilanguage software. Instead of software written in one language needing only to load its own resources, different parts written in different languages each need to be able to locate their own resources, and to load code and resources from parts written in other languages too.

5.3 Anti-pattern: Unverified Third-Party Software

Description

A package's build relies on third-party software, but does not probe at configuration time whether the required functionality works.

This anti-pattern is not about merely using third-party software, but assuming that it works, especially when there are known bugs or incompatibilities in some versions of the third-party software.

Consequences

The build can fail due to bugs or incompatibilities in the third-party software, while making it seem as if the original package is at fault.

Evidence

The problem of missing or incompatible versions of required third-party packages is a well-known problem with building open-source software [60, Ch. 15]. An attempt was made to sidestep this issue by having Ubuntu's package manager install all prerequisite packages, using release versions that Ubuntu had found to be mutually compatible; errors were encountered anyway.

The PDF documentation for `synopsis` does not build because of a bug in the third-party `xmlroff` tool.

There is an incompatibility between some syntax used in `gnat-gps`'s project files and the `gnatmake` tool used to build it. The `gnat-gps` package also has some inconsistencies between field names in C libraries that it uses and its Ada bindings for those libraries.

On Ubuntu 9.10, `axiom` does not build at all due to its reliance on broken functionality in GNU Common Lisp. GCL's `save-system` function does not work on that platform. This failure occurs despite the `axiom` source code distribution including the full source code for three different releases of GCL, and patches for a total of fourteen different releases.

Remedies

Functionality of third-party software can be verified at configuration time. Consider the approach that autotools uses for finding a C compiler. It tries a sequence of common compiler command names and compiler installation locations, and, for each compiler found, attempts to build and execute a test program written in C. If no functioning compiler is found, an error message is displayed that no working compiler is available and one must be supplied before the software can be built.

A similar verification at configuration time by `axiom` for GCL's `save-system` function would make it clear that GCL, not `axiom`, is the culprit, and the large amounts of time spent trying to debug the build could be spent updating GCL instead.

Each project will likely find it most efficient to probe for specific bugs that it has previously encountered in third-party software, rather than running general-purpose "Smoke Tests" [43], which will likely not check for the relevant bugs, or full regression-test suites for third-party software, which are generally too time-consuming to be used at configuration time.

Applicability

While this anti-pattern applies to all build systems, it may be more likely with multilanguage software due to use of tools such as GNU Common Lisp that are not used or updated nearly as often as more popular tools like the GCC C compiler. For example, as there have been no new releases for GCL on its homepage since 2005, many Lisp packages using GCL to perform low-level operations may want to verify at compile-time that the operations still function correctly.

5.4 Anti-pattern: Incorrect Dependencies

Description

Dependencies used by the build system are incorrect. Often the cause is manual specification of dependences where automated extraction tools could be used instead.

Consequences

When some dependencies are missing, artifacts that depend on other changed artifacts are not rebuilt, and the build products are not consistent with the source code. This can cause much confusion for developers who change source code and rebuild the package only to find the software's behaviour unchanged.

When extraneous dependencies are specified, build times can be much longer than they need to be.

Evidence

The header file dependencies for `python3.0` are manually specified, and for convenience, the same list of nearly all header files is declared to be a dependency of every implementation source file. The result is that almost any change to any header file forces a lengthy rebuild of all of `python3.0`, while changes to some header files that are not on the list do not result in any rebuilding. While this may be intended as a mechanism for detecting incompatible changes to Python header files, more precise dependency specifications would be as effective.

The automatically-generated build system for `ruby-prof` does not include any dependencies on header files. Changes to header filers are ignored by the build system. This is arguably a bug in the Ruby library that automatically generates the build system.

There is also a Windows binary library of the C part of `ruby-prof` included in its source distribution, as most Windows systems do not have compilers installed. However, there is no mechanism to keep this up-to-date with the source code when developing on non-Windows platforms, or to warn developers that they must use a Windows system or a cross-compiler to update the library artifact.

In contrast, `gnat-gps`—once it is built—has no problems with tracking dependencies in rebuilds. There are no C header files, only implementation files, in `gnat-gps`. The `gnatmake` tool handles dependencies in Ada code automatically. Incompatible changes between Ada and C code files are detected by the linker, but there are no build-level dependencies between them.

The `synopsis` package also has no issues with incorrect dependencies. Each parser written using C++ is independently built using an autotools-generated build system that invokes the makedepend tool to automatically extract dependencies.

Remedies

Use automatic dependency extraction tools, rather than manually specifying dependencies.

The use of automatic dependency extractors is already effectively a "solved problem." The issue of computing the optimal set of dependencies has received a great deal of attention in previous work [11, 57, 58, 65]. However, such problems are not serious issues for any of the case studies. The `python3.0` rebuild issue ⚠1 with `importdl.h`, described starting on page 42, is somewhat contrived. The `ruby-prof` issue is not so much a case of having incorrect dependencies as it is of not having any at all, due to missing features in an immature library.

Applicability

This issue is specific to build systems.

It was expected that dependencies between artifacts implemented in different programming languages might be an issue. However, no such issues were encountered, and considering the case of `gnat-gps` helps to explain why. Some Ada source files rely on functions implemented in C, but there is no build-level dependency between the source files, only a dependency of the final executable on intermediate objects independently derived from C or Ada sources. Intermediate object files produced by compiling C source files do not depend on Ada source files, or vice versa.

5.5 Anti-pattern: Ignored Error

Description

An individual build step fails in such a way that the build cannot succeed, but the build system continues without noticing the failure.

Consequences

If the failing build step does not result in subsequent detected build step failures, a corrupt build is produced.

Otherwise, the build eventually fails in a way that obscures the root cause. There are several other negative consequences. Time is spent waiting for unusable builds to finish. Time is spent trying to make sense of error messages that appear unconnected to the source of the error. Developers asking for help may be told something like, as is the case with `axiom`, “Your build failure originated much earlier in the log—could you please forward the entire thing?” [41]

Because the build system will have processed the build step as if it had succeeded, subsequent rebuilds may use stale or corrupt artifacts, producing an invalid build.

Evidence

The `axiom` build ignores the failure of build steps involving GCL, as discussed in that case study. The failures are inadvertently ignored due to a subtle error-handling interaction between Makefiles and shell scripts. Debugging the issue in order to discover the cause takes considerable time.

In contrast, build problems experienced with `synopsis` and `gnat-gps` are detected immediately by their build systems. The builds abort immediately with errors that, while also not straightforward to rectify, are at least closely related to their root causes.

The Makefile/shell script syntax issue that causes `axiom` to inadvertently ignore errors is also present in some `python3.0`, `synopsis`, and `gnat-gps` Makefiles. No build problems in the case studies resulted from this, but if some of the corresponding build steps fail, debugging the problem may be difficult.

Remedies

While this anti-pattern can occur with any build system, it is common when using Make, due to the subtle difference between commands separated by Makefile newlines and commands separated by the shell script '`;`' sequential list operator.

Uses of the shell script '`;`' sequential list operator should be replaced with the '`&&`' conditional operator. One popular Make textbook calls this "one of the most important aspects of programming robust Makefiles" [46, p. 236], yet many examples throughout the same textbook exhibit this anti-pattern.

One way to correct the error-handling behaviour for all commands in a Makefile, as well as to handle the additional cases of failing commands in pipelines and for loops, is to add the lines

```
SHELL = /bin/bash -eu -c 'set -o pipefail && eval "$${@}"'
.DELETE_ON_ERROR:
```

at the beginning of a Makefile, at least for GNU Make running on Unix. When intentionally ignoring errors, such as when `grep`ping for warnings in a log file, the shell idiom `||:`, which is shorthand for `|| true`, may be used.

However, these changes will not resolve all instances of this anti-pattern, even for Make-only systems, Some build tools, such as GCL when using it for its `save-system` function, do not report errors as nonzero return codes back to the shell, and additional tool-specific remedies are required.

Applicability

Correct error-handling is a general software development issue. However, the issue of inadvertently ignored errors is especially common in Make-based build systems. It is not specific to building multilanguage software.

5.6 Pattern: Build-Free Extensibility

Description

A package provides extension, modification, and adaptation mechanisms that do not involve compilation or build systems.

Consequences

End-users and developers can develop new features and customize the package regardless of how error-prone the build system is, and without needing specialized knowledge of build systems and their unique problems. However, the package itself and its extensibility mechanisms must still be built.

Evidence

The `python3.0` and `gnat-gps` packages can be extended with Python scripts that have full access to the internal APIs of each package, without building any C or Ada source code.

All of the problems with `axiom`'s build system matter little to most end-users and developers developing mathematics and mathematical algorithms rather than the internals of `axiom`'s interpreter-compiler. As long as they are able to get binaries from somewhere, they can develop freely inside the interactive `axiom` environment without dealing with the build system.

The `synopsis` and `ruby-prof` packages are implemented primarily as object-oriented packages in the dynamic languages Python and Ruby, respectively. While not all functionality is exposed as publicly-usable classes, it is possible to dynamically extend some of their functionality by subclassing.

Remedies

Packages that do not exhibit this pattern can integrate scripting support, using either general-purpose scripting languages such as Python, or by implementing custom languages such as Scratchpad for specialized domains.

Packages that are already written in dynamic object-oriented languages can make portions of their internal APIs public.

Applicability

This pattern is specific to build systems. It is not a multilanguage issue *per se*, but adding a scripting language to an existing package can turn an existing package into a multilanguage one. Once a scripting interface exists, it seems natural to start using it to implement some features.

5.7 Pattern: Object-Oriented Builds

Description

The build system can be dynamically customized and extended using object-oriented APIs by both project developers and by third-party build libraries.

Consequences

Object-oriented build systems can provide build systems with the benefits of encapsulation and reuse familiar from other object-oriented software. When bug fixes and new features are developed for build systems, they can automatically propagate to many projects.

However, object-oriented build systems may not be as mature as tools like Make, and may be buggy or lacking features.

Evidence

The `python3.0` project uses the `distutils` library to build its standard library. No problems were encountered with Python's build system for the parts where `distutils` was not used. There were, however, some build problems with the interpreter which is not built by `distutils`. Measured by lines of code, the problematic part not built by `distutils` is only one-quarter the size of the part built by `distutils`, and has less complicated configuration and construction requirements. As well, `distutils` is a well-tested standard tool used to build other Python-language projects, such as `synopsis`.

The Rake tool used by `ruby-prof` enables `ruby-prof` to export functionality for profiling unit-tests at build time as a general-purpose library that can be loaded by other Rake-based build systems. However, some of the problems encountered with building `ruby-prof` are related to the use of object-oriented builds: the questionable design decision to defer building until install-time, and the bug that omits header file dependencies from generated Makefiles, are symptoms of an immature ecosystem.

Remedies

Use a modern object-oriented build tool such as Rake, `distutils`, or SCons. Packages that provide useful build-time functionality can export that functionality as general-purpose build system modules. For example, a profiler could export the ability to profile performance-cortical unit tests at build time, for use in the build systems of applications such as web browsers or video encoders.

Applicability

This pattern is specific to build systems, and applies to both single-language and multi-language software. Just as object-orientation allows implementations to better deal with increased complexity, it is possible that object-orientation can help build systems deal with the additional complexity of building multilanguage software.

5.8 Pattern: Persistent Configuration

Description

A complete configuration is determined before starting the construction phase, and that configuration data is persisted in some way, for example as a file on disk.

This pattern is not at odds with dynamic run-time configuration, but addresses only those configuration parameters that are decided at build time.

Consequences

A persistent configuration allows for consistent builds in which each build step uses the same configuration data. This eases debugging of the build system, as post-configuration build steps become independent of one another, rather than relying on each other for configuration decisions.

There can, however, be some loss of modularity when builds of individual components must rely on an external configuration process.

Evidence

The Python console does not work in `gnat-gps` due to a configuration conflict, in which the Python library version used by the linker is hard-coded as 2.5, while compilation uses a different Python header version dynamically determined during the construction phase. This could be resolved by configuring the Python version only once.

Although `gnat-gps` does use an autoconf-based persistent configuration for many parameters, some system library paths determined by autoconf are not used by other parts of the build system, causing compilations to fail.

It is difficult to manually rerun `axiom` build steps because configuration is dynamically determined in top-level Makefiles and passed down as environment variables to subsidiary Makefiles in subdirectories. Build steps in subsidiary Makefiles can only be invoked correctly via limited interfaces of top-level Makefiles, and debugging attempts to manually invoke individual build steps fail due to missing configuration data.

Part of `ruby-prof`'s build system configuration suffers from excessive persistence: configuration data, including compiler options, from when the original Ruby interpreter was built is used. This precludes the use of binary-compatible C compilers with different command-line option syntax.

In contrast, another part of `ruby-prof`'s build system, implemented by core Ruby libraries and used without modification by `ruby-prof`, suffers from excessive persistence. `ruby-prof` has some optional profiling features that are only functional when special instrumentation code has been compiled into the Ruby runtime. Although install-time configuration probes are made for these features and are used to conditionally compile C code, the results of these configuration probes is not available to the main `ruby-prof` executable. The `ruby-prof` executable nevertheless documents and accepts command-line options for enabling optional features which have not been compiled, causing `ruby-prof` to crash.

The two remaining case studies, `python3.0` and `synopsis`, both use autoconf to determine persistent configurations for building their C/C++ source code, and problems similar to those of other packages are not encountered in the case studies.

Remedies

While not necessarily using the autoconf implementation, use the design decisions that autoconf gets right [66, 74]: perform configuration probes as an initial phase separate from construction, and persist the probe results to be used throughout the build.

If preserving modularity is a concern, the build system can be modified to automatically generate an initial persistent configuration for multiple components.

Applicability

This pattern is specific to build systems. It is not specific to multilanguage software. However, lack of mature configuration tools for some languages may be a factor.

5.9 Discussion

This section discusses the significance, generalizability, uses, and limitations of build patterns and anti-patterns.

The significance of the build patterns and anti-patterns is that, because they are inferred from a comparative study of unrelated software packages, they represent shared problems among build systems for multilanguage software. The implication is that build systems for multilanguage software are error-prone, but problems among them could be systematically addressed due to the commonalities among the problems. All but one (anti-)pattern—Filename Collision—is supported by observations from multiple case studies. The only 'one-off' issues that could not be synthesized into patterns or anti-patterns were (1) `axiom`'s absence of support for incremental rebuilds, and (2) that the Ubuntu maintainer for `gnat-gps` rewrote the build system. Otherwise, the build issues encountered in the case studies are instances of general build problems, for which general solutions could be found. Even the patterns for which only one observation was problematic, namely Filename Collision and Ignored Error, could be issues for other packages; they are not highly localized to one specific package and do not require *ad hoc* solutions.

While these patterns and anti-patterns can be used by practitioners who are developing individual build systems, they are intended to be used in a more general way. Specifically, they are intended to be applied in the design and implementation of build system frameworks and tools. Build patterns can be designed into such tools, and build anti-patterns can be used to guide the development inside build system tools and frameworks of automated detectors and correctors for classes of build system problems.

Possible uses and prospects for automation vary among (anti-)patterns.

- Filename Collision detection is easily automated for straightforward case collision. However, an effective tool may require a comprehensive database of various filesystem filename legality rules and potential programming-language-specific filename conflicts, such as Python's potential conflict between the module directory `Foo/` and the file `foo.py`. As well, developer input is required to supply meaningful replacement filenames.

- Incorrect Dependencies is systematically correctable by using automated tools such as dependency extractors. Such tools can be used automatically by build system frameworks, and automatic dependency extraction can be built in to compilation tools.

- Ignored Error is largely amenable to automation. Error-handling could be emphasized as a requirement in the design of build frameworks. However, not all build tools report error in standard ways such by returning appropriate process exit codes. They may write error messages only to the terminal or a log file, or they may create zero-sized or otherwise corrupt output files. A database of of tools known to conceal errors, and executable techniques for detecting these errors anyway, such as with regular expressions to check log files, could be useful to many build tools.

  For legacy Make-based build systems, automated tools could fix problematic syntax such as use of the '`;`' shell operator. Since a valid need to ignore errors from some build tools does sometimes exist, syntax such as the '`-`' prefix [21] can be used inside Makefiles, both to ignore errors and to declare to an automated tool that error suppression is intended.

- The use of Persistent Configuration is not something that could be easily determined by an automated tool. It is instead a valuable technique from the design of autotools, and it is called out because new build tools and frameworks may not understand or value its importance.

- While the use of Unverified Third-Party Software can be done in a general way to some extent, for example when autotools probes to find a working C compiler, it is most effective when used on a case-by-case project-specific basis. For example, where `axiom` knows that there are build problems with some versions of GCL, it can provide probes for the known problems.

- Implementing solutions to Installation Required can be automated to some extent by providing appropriately-designed templates for new projects that some object-oriented build tools can generate. It can also be addressed in a general way through the use of standard libraries for managing search paths, such as the `pkg_resources` package [1] that accompanies a modified version of `distutils`. This particular path management library can also help avoid the Installation Required anti-pattern by being integrated with the build system.

As discussed under the Remedies for Installation Required on page 80, virtual machines and virtual-machine-like environments can help mitigate some negative consequences of this anti-pattern, but such mechanisms do not address the key negative consequence of the anti-pattern.

There are certainly existing software packages with this anti-pattern that could not be resolved without significant rearchitecting of the implementation.

- Apart from its helpful effect on modularity of build code, the use of Object-Oriented Builds can have a synergistic effect to reduce the effort required to use patterns and address anti-patterns. Detectors such as that for Filename Collision could be implemented as modules to be imported into build systems with a few lines of code, as `ruby-prof`'s build-time `ProfileTask` functionality can be. Additionally, practitioners encountering their own build problems in their own software, rather than in an academic case study, can develop their solutions in general-purpose ways and distribute them for use by others. With Rake, for example, it is already possible to distribute reusable build actions via Ruby's extensive existing web-based infrastructure for curating and distributing third-party libraries.

  Note that while the popular Apache Ant tool has support for plugins, and dozens of mature plugins are available for it, it does not fit the Object-Oriented Builds pattern because Ant-based build systems cannot dynamically extend or modify themselves. Ant build systems are implemented in a non-general-purpose programming language and do not have access to the Ant plugin API. Ant plugins are developed completely separately from and in a different language than the build systems. There is no evolutionary path by which build functionality can initially be developed as part of a build script and then, once sufficiently mature, be released as a plugin.

- Build-Free Extensibility is more of a feature that a software project must architect and design, rather than something that can be had for free by using the right build tool or build framework. However, it seems to be a worthwhile feature to implement for software that grows large, is intrinsically difficult to build, or has strong interactivity requirements, among other possible reasons.

  It is possible that building multilanguage systems is simply more difficult than building single-language systems, and that while research and development of multilanguage

[1] http://packages.python.org/distribute/pkg_resources.html#resourcemanager-api

build systems can reduce the gap, it cannot eliminate it. If this is the case, effort may be better spent on improving run-time extensibility rather than on the build system. The Build-Free Extensibility pattern can shift the build difficulties away from end-users and casual developers to build experts. Projects could accept that building their software requires expert knowledge and encourage users to use binaries if possible, while still allowing for a great deal of customization and extension by non-build-system experts.

What are the limitations of patterns and anti-patterns? While they could be useful to practitioners, they take effort to implement or to resolve, both in build tools and frameworks and in individual build systems. Individual projects may have their own differing priorities that conflict with using build patterns or resolving build anti-patterns. A software project may feel that its build system works fine, is tangential to the goals of the project, and does not require any additional development. Or, for example, it is possible that a Unix software project, in addition to not resolving a Filename Collision, may actively disdain Windows users[1] requesting such a change. Automated support in build frameworks and tools could reduce the burden of addressing (anti-)patterns for individual projects.

Some (anti-)patterns may not be applicable in all environments. For example, regarding Ignored Error, Humble and Farley [33] caution that errors *should* be ignored inside a unit test suite that runs as part of a build: if there are multiple failing unit tests, developers need to know about all of them, not just the first one, to create a proper fix. However, if every single unit test is failing, there is likely a more serious problem, and aborting the tests quickly in such a case may be appropriate. Another example, regarding Persistent Configuration, is that the influential *Twelve-Factor App* set of best practices for deploying web applications [71], written by the Chief Technology Officer of the Heroku cloud webapp platform, advises storing configuration exclusively in environment variables—the opposite of the conclusion drawn in this chapter based on the `axiom` case study. The difference is that Heroku's priority is to use the simplest possible interface between its platform and the applications that run on it, in order to allow both to evolve independently. That priority outweighs the concerns associated with the anti-pattern.

In conclusion, build patterns and anti-patterns are significant because they represent commonalities among the diverse build systems in the case studies. They are intended to be used in generalized settings such as build frameworks and tools. However, their use will not necessarily be appropriate in every case, and they may be considered to be of low priority by individual projects.

[1] http://dilbert.com/strips/comic/1995-06-24

Chapter 6

Discussion

This chapter discusses the significance and implications of the results. First, previous discussion sections are revisited. Then, possible causes for build problems are examined, and the phenomenon of leaking abstractions is proposed as a partial explanation. Effort and tool requirements for understanding build systems are discussed next. The chapter ends with discussion of threats to validity and related work.

There are two other discussion sections in this thesis: one for the procedure used to select the case studies, and one for the build patterns and anti-patterns.

The discussion section for the case study selection procedure, Section 3.3 starting on page 21, discusses the diversity of the results and potential threats to validity of the procedure used to select the case study systems. That discussion left open the question of whether the case study selection was suitable. That question can now be answered. As this is a qualitative case study using analytic inference, evaluation of the selection is determined by examining whether the case studies are sufficiently interesting and diverse that useful comparative conclusions can be drawn [73]. The commonalities expressed in the form of the build patterns and anti-patterns of Chapter 5, and the implications discussed in this chapter, are evidence that the selection was indeed suitable.

The discussion of the build patterns and anti-patterns, Section 5.9 beginning on page 89, examines the significance, generalizability, uses, and limitations of the build pattern and anti-patterns in the context of developing build systems, tools, and frameworks that leverage the patterns and avoid the anti-patterns.

This chapter examines the case studies and the build patterns and anti-patterns in a broader context. The most significant result is that four of the five case studies had build problems preventing an initial build from succeeding, and the fifth case study had rebuild problems. As the case study systems are diverse along a variety of dimensions, as detailed in Section 3.3, were systematically selected without knowing ahead of time how they would build, and exhibit commonalities in their build problems, it is concluded that build systems for multilanguage software are error-prone; that there are commonalities in their build problems; but that these could be systematically addressed.

The research questions set out in the introduction are as follows:

RQ1) What are the major issues in building multilanguage software?

RQ2) How can these build issues be addressed?

RQ3) Why do these build issues occur?

This thesis proposes build patterns and anti-patterns as a way to address these questions. The fact that there were so many issues in getting the case studies to build at all provides one answer to RQ1: the major issue in building multilanguage software is getting it to build at all. This is a significant finding in light of the fact that previous work on single-language build systems focuses on improving the convenience, correctness, and/or speed of existing already-working build systems [34, 60]. The build patterns and anti-patterns themselves provide another answer to RQ1, in that they specifically identify some of the major issues. RQ2 is also addressed by the build patterns and anti-patterns themselves. This chapter discusses RQ3.

6.1 Build problem causes

Why do build issues occur? There will always be bugs, but what causes underlie the build problems represented by the build patterns and anti-patterns? Some (anti-)patterns represent requirements that may rarely be made explicit, and therefore simply do not get implemented unintentionally. Installation Required and Filename Collision are examples of this: developers are unlikely to value making software run from the build directory, or testing it on unsupported platforms, when that time could be spent improving the deliverable product. Other (anti-)patterns appear to arise from lack of knowledge of specialized tools and techniques for build systems: Incorrect Dependencies, Persistent Configuration, and Unverified Third-Party Software. However, knowledge of specialized tools and techniques for build systems is of limited value in improving the user experience once the software is installed. Developers omitting these build system requirements from their packages, and lacking specialized build-system knowledge, are not lazy or ignorant: they are being efficient by spending time on development and skills that contribute directly to the product being created and the associated value being delivered, rather than the somewhat tangential topic of build systems.

That said, two patterns hold promise for improving build systems regardless. Object-Oriented Builds allows developers to develop their build systems in general-purpose languages, using encapsulation mechanisms that developers are already deeply familiar with. This is in contrast to, for example, the shell script, Makefile, and M4 macro languages that comprise an autotools-based build system which have extremely limited modularity and encapsulation abilities. Modules that automatically resolve build anti-patterns could be loaded into object-oriented build systems. Prospects for reuse also mean that developers may find it easier to justify working on their build systems, and to solve their build problems in general-purpose ways. Embodying the solution to a build problem as a general-purpose build system plugin not only solves the problem now, but can solve the same problem for members of product families within an organization, and can be publicly release to solve the same problem for many different build systems used by many different projects. It is also likely more appealing to developers to develop reusable modules than to spend time debugging a hacked-up shell script embedded inside a Makefile, the result of which will only ever be usable by a single build system. The ability to solve problems in general ways could reduce the view of build systems as 'tangential.'

The other promising pattern is Build-Free Extensibility. While it also has the potential to lead to essentially unbuildable packages, adding scriptability to a software package involves implementing similar features to that of a build system. The shared goal of a build system and a build-free software extension interface interface is to turn source code into running code. When extensive features are implemented using build-free techniques, the host package becomes responsible for correctly dealing with issues such as setting search paths correctly, and such as reloading changed scripts, even if they are only indirectly referenced. Once this is part of a key software feature, namely the extension interface, it is no longer tangential. A concrete example: `python3.0` has internal routines for loading referenced script modules that are carefully thought out and tested. The build system does not and cannot receive the same attention. Moving build code directly into the application may be the answer. The build system already takes on abstractions from the implementation and application domains anyway.

6.2 Leaking abstractions

One interesting finding of this thesis that may help explain RQ3 about why build problems arise is the observation that abstractions, mental models, and other concerns from the application and implementation domains may 'leak' into the build system. That is, the build system has properties that resemble, at an abstract level, properties of the software that are not related to the build system, and whose presence may only be explainable by reference to the application and implementation domains.

Specifically:

- `synopsis` uses independently-developed and -built third-party components that are integrated by using a standardized interface. The only major issues are library search path setup issues.

  This approach is similar to the architecture of the experimental CORBA-based windowing system for which `synopsis` was developed, wherein independently-developed and -built components communicate using standardized interfaces. Service lookup and path configuration are common problems in getting CORBA software to run.

- `python3.0`'s build structure uses the design philosophies of `python3.0` itself: the build system is simple, flat and explicit [53]. Like `python3.0`, the build system has a relatively small core but is still powerful. However, since explicit lists of dependencies are used rather than 'magic' dependency extractors, build system rebuild problems result. The desire to be explicit and avoid 'magic' is common to both `python3.0` and its build system.

- `gnat-gps`'s AdaCore build system has an extremely complicated hierarchy of sub-builds, as shown in the upper portion of Figure 4.11 on page 49. The build system is sufficiently difficult to get working that the Ubuntu maintainer for the package rewrote the build system, citing the "complex structure of project files importing and including each other" and "evil recursive Makefiles" as reasons to rewrite it in the source code of the new system.

  However, the build structure makes sense in terms of the goals of the developers. `gnat-gps` is a product of the AdaCore corporation, a commercial Ada vendor. They provide free, open-source compilers and tools to individuals, while selling support and products such as safety-critical offerings to other corporations. AdaCore the corporation uses `gnat-gps` as a development environment for Ada-language components that start out as parts of `gnat-gps` but are released as separate libraries once they mature. Examples include the GtkAda window-system bindings, and the GNATCOLL component collection which implements features such as logging and interfacing with scripting languages. These libraries are released with the intention of making Ada a more attractive language for development, thereby boosting sales. The hierarchical build supports this strategy by ensuring that components do not become inadvertently coupled to other parts of `gnat-gps`, which would make it harder to release components as independent packages.

- `axiom` presents the user with an interactive 'Scratchpad' environment for doing mathematical work and developing mathematical algorithms. This environment is highly dynamic, and allows new mathematical objects to be defined and redefined.

  However, building `axiom` is a complicated process; it is not even possible to build it on the version of Ubuntu used in this study. Even when `axiom` can be built, changing the implementation code for the Scratchpad interpreter-compiler entails a complicated process of marking up literate source code in a LaTeX-based macro language, completing a lengthy rebuild of the software from scratch, and running comprehensive regression tests to ensure correctness. There is no support for incremental rebuilds.

  One possible parallel for this unusual build system is the process by which some professional mathematicians work. Concepts and proofs are initially developed on scratch pads of paper. When a generally useful result is obtained, adding it to the mathematical literature becomes a time-consuming process. The result must be carefully written up—often using LaTeX—and submitted to a lengthy peer review process to ensure correctness before it is finally published. This process mirrors that of `axiom`'s build system.

- `ruby-prof`'s build system is largely implemented by external libraries. The build system is adequate, and has novel and potentially useful features, but is not without problems. However, the build problems are largely bugs in standard Ruby libraries, and are likely to be fixed by library upgrades. Some of the bugs encountered during the case study have been addressed in more recent releases. The condition of the build system mirrors the developer-oriented Ruby ecosystem, where experimentation is valued, and it is expected that bugs will be fixed by frequent library upgrades [72].

While leaking abstractions may have negative effects on buildability, it is not necessarily always good or bad. It seems to have clearly negative effects on the buildability of `axiom` and `gnat-gps`. However, the effect on the other case studies is mixed, as both advantages and disadvantages of the overlying abstractions propagate to the build system.

`python3.0` has a small, simple, and effective build system, which occasionally causes build problems by avoiding 'magic.' `synopsis`'s build system exploits many disparate components through simple interfaces, but it can be difficult to configure the components to find each other and work together properly. `ruby-prof`'s build system relies on libraries with useful novel features, but these somewhat experimental libraries also have bugs.

If negative effects are more likely, this could be relevant towards McIntosh's conjecture that concentrated build system ownership, wherein a small team of build system experts maintain the build system, results in lower build maintenance effort than having build maintenance dispersed among the entire development team [45]. The implication from this thesis would be that build system experts working primarily on the build system, and having limited exposure to the application and implementation domains, are less likely to incorporate application and implementation domain abstractions into the build system where they may not be appropriate.

This finding is highly speculative, but in some sense not all that surprising: people use what they already know when attempting new tasks. Further research could shed light on the topic. One possible approach is a systematic study comparing the various vocabularies

Package	Time to first successful build	Time to understand build system
`synopsis`	< 1 hour	1 day
`python3.0`	< 10 minutes	a few days
`gnat-gps`	10 hours before deciding it is unbuildable and switching to the Brenta build system; then 2 full days of debugging to resolve build problem [7]	time to learn Ada, `gnatmake` basics + 1 day
`axiom`	10 hours before deciding it is unbuildable; then 10 hours of debugging to find the cause	weeks
`ruby-prof`	< 1 hour	time to learn Ruby, Rake basics + 1/2 day

Table 6.1: Subjective estimates of build effort

used in the application, implementation, and build system domains.

6.3 Effort and tool requirements

This section discusses the effort needed to build the case studies, to understand their build systems, and tool requirements that could improve the process.

Rough estimates of the time required to build and understand the case study packages are given in Table 6.1. These are inherently subjective estimates made long after the studies were actually conducted, and dependent on the author's prior experience. In this context, 'understand' is a mostly undefined term that means to be able to make small changes to the build system, and explain its high-level operation to someone else, but does not encompass the level of detailed knowledge required to, for example, make build diagrams similar to chose of Chapter 4 for the build system.

The estimates are based on the experience of the author who was already familiar with the Lisp, Python, C, and C++ programming languages, as well as autoconf and Make-based build systems, but had no prior experience developing with `distutils`, Ada, or Ruby.

The understanding time for `gnat-gps` is only 1 day because the compilation model is so similar to that of C. The understanding time for `ruby-prof`'s build system is even smaller because there are only a handful of source files.

The technique used to understand the build systems for the case studies was almost exclusively manual inspection of source code and documentation. There are exceptions for two case studies. The multilanguage architecture for `python3.0` in Figure 4.5 on page 39 was produced in part by using the ASX [15] and LSEdit [63] fact-extraction and software landscape browsing tools to extract and organize `python3.0`'s function-call architecture. For the `gnat-gps` case study, custom scripts were written to extract dependency information from the Ada `.ali` files generated by the compiler, and from the Makefiles, in order to generate graphs of the component hierarchy reminiscent of the work of Mason and Ochotta [42]. Incidentally, that paper is what first got me interested in doing software engineering research.

The major requirements that a tool to support the build system understanding process needs to have are:

- The ability to show different kinds of relationships between various software artifacts
- The ability to group different software artifacts together as one abstract entity

As understanding of the build systems in the case studies of this thesis was mainly a manual process, one extremely useful tool that supported these requirements was the OmniGraffle diagram editor available from http://www.omnigroup.com/products/omnigraffle/. It is a vector-based diagramming program that allows labeled shapes to be drawn, connected, and moved around. It also embeds the Graphviz toolkit for graph visualization [24], which was extremely useful for Figure 4.10 and Figure 4.11.

Development of the diagrams was an iterative process. As hypotheses were developed about how the multilanguage software was architected and how the build system works, the tentative hypotheses were diagrammed. These hypotheses were verified, and parts of the diagram that required additional understanding were investigated, using techniques such as examining documentation, examining source code, and running experiments involving the build system.

To support the evolution of the notation that was developed and refined in concert with the diagrams, each graphical element was tagged with 'user data' denoting the artifact or relationship type and/or programming language. Changes to the graphical notation were applied by running a script that iterated over each graphical element and set the corresponding graphical style.

In hindsight, a more automated approach that also supports these requirements could have been developed by using standard fact extractors to determine build dependencies and runtime call structure between artifacts, manipulating the extracted facts using relational algebra to automatically group multiple artifacts into abstract entities, and exploring them using a visualization tool. A concrete implementation of such a system could consist of the LSEdit [63], ASX [15], and grok [69] tools, along with fact extractors for the other languages encountered, and newly-developed fact extractors for build systems. While such a project would be interesting future work, the process of understanding the case study build systems in this thesis was dominated by cognitive processes, such as learning the workings of and relationships between unfamiliar tools, libraries, and programming languages. The packages were sufficiently diverse that there was little overlap between the understanding activities across packages. For understanding similar software packages using similar languages and build technologies, automation would certainly be useful, but for this study, automation would have helped little.

6.4 Threats to validity

This section examines potential threats to the validity of the results of this thesis. As with the discussion of threats to the validity of the selection procedure in Section 3.4 on page 24, these threats are examined using four standard criteria [73]: construct validity, internal validity, external validity, and reliability.

Construct validity—are the various measures used measuring what they are intended to? The research goals of this thesis were to determine the major build issues for multilanguage software in general. However, instead of examining multilanguage software in general, Ubuntu was used, both as a source of case study packages and as a platform to attempt to build them on. The case study packages do not necessarily support the Ubuntu platform. However, as Ubuntu is extremely popular, many stakeholders would be using it, making it a valid host platform for builds. As for Ubuntu being the source from which packages were selected, that may limit the generalizability of the results to open-source Unix packages, but does not affect the validity of the results within that realm.

This issue with reliance on Ubuntu was addressed in part by using vendor source code distributions, not Ubuntu ones. Additionally, no Ubuntu-specific build problems were encountered, with the possible exception of `axiom`'s failure to build due to GCL incompatibilities.

Internal validity—are discovered relationships more than just coincidence? Many of the observations in the case studies are facts about the outputs of programs on known inputs, rather than possibly spurious inferences. Spurious relationships are avoided in the synthesis of build problems into patterns and anti-patterns by the use of comparative analysis, which requires multiple data points for each inference. As for the discussion about how build problems arise, that is admittedly speculative, but not out of place in an exploratory study—exploratory studies are conducted to guide future research that can clarify the potential issues and conjectures raised in the exploratory phase.

External validity—how generalizable are the findings? This thesis examines five case study packages, and analyzes the commonalities among their build problems. It may seem possible that the build patterns and anti-patterns could apply only to these five systems, and nothing else. However, the research design uses analytic inference, not statistical inference. For example, Installation Required is a real problem in real build systems because it was encountered as a problem in several diverse build systems, and `python3.0` had special-purpose code to avoid the anti-pattern. This could not be the case if these were one-off problems, or problems limited to only these five systems. So while these findings are generalizable to similar software packages, the generalizability can only be quantified by quantitative studies.

Additionally, this thesis examines only open-source Unix software. The selection is diverse along many dimensions but still consists largely of programming language implementations and tools. Whether that reflects the nature of Ubuntu software, or is the product of bias in the selection procedure, or results from some other cause, is not known. From the perspective of examining the packages investigated in the study, it is not known to what degree the build patterns and anti-patterns can be generalized beyond the realm of open-source Unix programming language implementation and tool packages. However, considering the build patterns and anti-patterns produced by the study, none appear relevant to only such packages.

Reliability—could someone else do this and get the same result? There are simply too many inferences involved in this study that rely on the individual knowledge and experience of the investigator for anyone else to get exactly the same result. However, someone else conducting the same study would get a similar set of build problems, and would synthesize them into a mostly similar set of (anti-)patterns, and interesting discussions could ensue about disagreements of the interpretation. This claim is substantiated by the fact that Smith, in his textbook on build systems [60], produces a much longer list of recommendations for build systems that have much overlap, but some of which the investigator of this study would disagree with based on the outcomes of these case studies; for example, Smith's recommendation to "Use a Consistent Build System" precludes successful heterogeneous build systems such as `synopsis`'s `distutils`–autotools build.

While the selection of packages is completely reproducible, as well as the test build environment down to the first error message from the build system for each package, at that point the sole investigator must rely on his or her judgement, experience, and knowledge in order to correct the build problem and move on to the next one.

This threat to validity is addressed somewhat by random selection, in that the investigator encounters unfamiliar packages using unfamiliar programming languages, where prior knowledge and prejudices are less applicable. However, familiarity with some of the languages in the case study can colour interpretation of build problems. For example, the way that Ruby's build system for extensions does not build them until install time, even for development versions, seems strange and problematic to someone with a Python background, but Python's methods may seem strange and problematic to someone from a Ruby background. The case study systems used several languages completely unfamiliar to the investigator. The investigator had not heard of any of the packages but `python3.0` before conducting this study, and, although quite familiar with the Python language, the investigator was not familiar with `python3.0`'s software architecture or build structure. However, the presence of `python3.0` in the majority of the packages may have affected the results of the study. Purposive selection of cases that includes important dimensions such as investigator familiarity with programming languages, and thinking through possible threats to validity while designing the study but before conducting it, are recommended case study guidelines [73]. Another common way to ensure reliability in case studies is to have multiple investigators examine the same cases and then compare their results. However, that is likely too heavyweight for an exploratory study such as this one.

6.5 Related work

In his doctoral dissertation, Adams [2] performs deep case studies of the build systems of a number of packages, including the Linux kernel [5], Quake, and a number of industrial packages. He develops an interactive open-source tool called MAKAO [4] for visualizing and refactoring Make-based build systems, and for integrating aspect-orientation, a form of multilanguage programming, into build systems for legacy C software. While this thesis also performs deep case studies, its goal is to understand build problems and their causes, while the work of Adams aims to understand how and why build systems change in response to implementation source code changes, and how these co-evolving changes can be managed [3]. Although this thesis's visualizations are manually created while MAKAO automatically produces its diagrams automatically, this thesis goes beyond MAKAO by examining multilanguage software that uses a variety of build systems instead of only C software built by Make, and abstracting away implementation details to form conceptual build architectures. Adams also discusses build 'patterns' and 'idioms' that are common methods of implementing desired build functionality despite the limitations of Make, such as the circular dependency chain pattern used throughout the build system of the Linux kernel [2, Figure 4.15]. In contrast, the (anti-)patterns of this thesis involve higher-level functional concerns, not their implementation.

Suvorov et al. [62] perform case studies of several large-scale build system migrations, such as when the KDE project migrated its build system to CMake from autotools [49]. This thesis aims to understand build problems and their causes in multilanguage software, while Suvorov et al. analyze the techniques of and forces behind successful and unsuccessful build system migrations.

Godfrey and Tu introduce build-time software architectural views [67], and discuss build-system architectural styles such as the "code robot," which is an auxiliary tool that is built and executed in order to automatically generate platform-specific source code, enabling highly-portable software. The notation for the diagrams in this thesis are based on their work. The diagrams in this thesis differ in that they attempt to show a more abstract view of the build, by using larger groupings of components, and by abstracting away many build implementation details to make the structure more apparent. The diagrams in this thesis show build problems in the diagrams as well, while the work of Godfrey and Tu deals with functioning builds. The "code robot" build-system architectural style that they document could be described in the form of a pattern analogous to the Build-Free Extensibility pattern of this thesis; McIntosh refers to this build-system architectural style as a pattern [44].

McIntosh et al. empirically analyze the revision history of ten projects in order to estimate the overhead associated with build maintenance [45]. They find a wide disparity in estimates that could be caused by different build code ownership styles: a team of build experts solely responsible for builds, or all developers sharing maintenance on the builds. They conjecture that centralized build ownership may be more appropriate for open-source systems. This thesis uses qualitative techniques on a different set of software packages, and finds evidence of application- and implementation-domain abstractions 'leaking' into the build domain, a phenomenon that could help explain reduced overhead and improved buildability for centrally-owned builds. In his master's thesis [44], McIntosh describes these different ownership styles as organizational patterns among developers, while the patterns of this thesis address build system structure and implementation.

Tamrawi et al. develop static analysis techniques for Makefile source code that permit analysis and refactoring [64]. This thesis examines a variety of different build systems at runtime. Their techniques are also used for the static detection of Makefile "code smells," which are conditions such as recursively-defined variables or cyclic dependencies that cause Make warnings or errors at build time. None of the build problems encountered in the case studies of this thesis involved such Makefile "code smells."

Miller's article "Recursive Make Considered Harmful" discusses a number of problems with slow or incorrect builds that the author has experienced, and attributes them to the use of recursive invocations of Make [47]. The author shows how the problems can be overcome by an improved build system structure. The same strategy is used in this thesis, where commonalities among build problems in various projects are analyzed and labeled as symptoms of anti-patterns, and corresponding remedies are proposed. This thesis examines a variety of build systems using a variety of programming languages, and refers to specific systematically-selected projects.

Smith's practitioner-oriented textbook on build systems has an extensive discussion of "Various Ways to Reduce Complexity" [60, Ch. 17]. Some of his recommendations, such as "use a modern build tool," "automatically detect dependencies," and "abort the build after the first error" match the recommendations of some (anti-)patterns. The (anti-)patterns in this thesis were developed independently, are empirically traceable to case studies, and are structured in the form of patterns and anti-patterns. Additionally, Smith's book is aimed at practitioners wishing to improve individual build systems, while this thesis explores the possibility of systematically addressing build system problems.

The practitioner textbook *Continuous Delivery* [33] provides an overview of modern build tools and build system practices, and expounds the needs for full automation. The related practitioner textbook *Continuous Integration* [18] discusses techniques for creating fully-automated builds within an organization, and why this is a good idea. One co-author has published a developer article about “build smells,” which are suggestions for improving the maintainability of build code for Java build tools [17]. Examples of such build smells include duplicated code in build scripts, and overly-long build script sections. There is no overlap between these “build smells” and the (anti-)patterns in this thesis.

Spolsky’s “Law of Leaky Abstractions” [61] bears some resemblance to this thesis’s findings on leaking abstractions. Spolsky’s law states that developers must be familiar with implementation details underlying abstract interfaces because abstract interfaces will sometimes ‘leak’ in the form of bugs from the implementation bubbling up. The existence of bugs which occasionally require source code written in high-level languages to be debugged at the assembly-code level is one such example. In contrast, the ‘leaking’ in this this thesis corresponds to the findings that when developers work in multiple domains simultaneously, abstractions from one domain may be reused in another, even if the domains are only loosely related.

Chapter 7

Conclusion

This thesis has shown that build systems for multilanguage software are not the "solved problem" they may appear to be. In fact, build systems for multilanguage software are error-prone. However, there are commonalities among the problems, allowing them to be systematically addressed.

The technique used was that of an exploratory qualitative study. Case studies of the build systems of five open-source multilanguage software packages were performed. The collection of five packages was diverse among a number of important dimensions shown in Table 3.8 on page 21, and the packages were systematically selected without knowing ahead of time how they would build. Four of the five packages had build problems which required manual intervention to get a running development version from source code. The fifth package also had problems rebuilding. The build problems experienced and the problem avoidance features observed throughout the case studies were synthesized into build patterns and anti-patterns using a comparative analysis. The resulting build patterns and anti-patterns are largely not specific to multilanguage software.

The research questions spelled out in the introduction are as follows:

RQ1) What are the major issues in building multilanguage software?

RQ2) How can these build issues be addressed?

RQ3) Why do these build issues occur?

One answer to RQ1 is that the major issue in building multilanguage software is getting the software to build at all, rather than the issues of convenience, correctness, and/or speed that previous work in this area has focused on [34]. RQ1 and RQ2 are also answered by the set of build patterns and anti-patterns, which summarize and provide remedies for the major build problems encountered. RQ3 is addressed in Chapter 6. It appears that build problems arise for two interrelated reasons: build systems are viewed as tangential to software project goals, so receive little attention; at the same time, the development that is done on build systems leaks abstractions from application and implementation domains into the build system, with mixed results. However, application of the build patterns Object-Oriented

Builds and Build-Free Extensibility may work together to help address these causes.

Along the way, some side contributions of this thesis include: a procedure for identifying multilanguage packages, which could be of use for future researchers; deep exploration of the build and multilanguage architecture along with observations of build problems and features for five case study packages; corresponding diagrams of the build and multilanguage architectures of these packages; tool requirements for conducting future similar studies; and a set of build patterns and anti-patterns with remedies.

7.1 Future work

The results of this thesis suggest a number of directions for future research on build systems. One of the central issues to be investigated is what factors contribute to whether software builds "out of the box" or not.

One natural question suggested by the qualitative exploratory study conducted in this thesis is the prevalence of the build (anti-)patterns. While they are present in the case study systems, how frequent are they in larger populations of software packages? The qualitative research method used in this thesis cannot answer this question directly. A large-scale quantitative study of software portability could help to quantify the degree to which build systems for multilanguage software are not a "solved problem." Related questions include whether specific programming languages are more likely to be associated with build problems, and whether there is a correlation with other software package dimensions such as age, software size, team size, presence of corporate sponsorship for a package, or package popularity. Correlations of build problems with code quality metrics could shed light on whether build problems are symptomatic of poor software quality in a package, or whether buildability suffers at the expense of implementation software quality. Since each software package is made automatically buildable by operating system distribution maintainers, metrics of maintainer effort, such as those extracted from the Ubuntu `.diff.gz` patch files distributed alongside the source code, may be useful. Studies could include other operating systems and operating system distributions to determine if buildability of the same software varies significantly between platforms. Additionally, evolutionary studies could examine the question of whether easily-buildable software has an easier time gaining users, gaining developers, and becoming popular.

Future work involving automated ways to implement patterns and to remedy anti-patterns, both for individual packages and as part of build frameworks, is discussed in Section 5.9. For many individual (anti-)patterns, such as Filename Collision, the future required work for automation consists largely of development work, not research. However, there are research questions suggested by some of the build patterns and anti-patterns. A better understanding of the Object-Oriented Builds pattern and its prospects alluded to in the discussion could result from, for example, case studies of existing packages that provide or use third-party build plugins. The Build-Free Extensibility pattern suggests empirical evolution studies about the effect that adding build-free extensibility mechanisms such as scripting support has on package popularity, and on the use and degree of integration of multiple programming languages in the software. Filename Collision also suggests studies of portability issues in open-source software. For example, the Python language is well-supported on many platforms, yet software such as `synopsis` written in it can have portability issues for reasons such as the use of C-language extensions, which can be expected to require some effort to be portable, but also for making assumptions about the underlying operating system such as what filenames are legal. How can a portable language such as Python encourage software written in it to also be portable?

Some of the manual techniques used to understand the build systems in the case studies could be automated, and could also help developers understand their build systems. Existing fact-extraction and analysis tools could be extended to support this. Of particular interest is the use of dynamic build tracing to automatically produce abstract build diagrams using the notation of this thesis, perhaps by extending MAKAO [4].

The finding of abstraction "leakage" is highly speculative and requires systematic study. Comparative studies of developer vocabulary between application, implementation, and build system domains, especially as they change over time, could help to understand this phenomenon. It would be interesting to understand what additional factors cause it to have negative instead of positive effects on buildability, or vice versa, and also to compare results for various build code ownership styles [44, 45].

Bibliography

[1] AdaCore. *GNAT User's Guide*, 4.3.4 edition, 2008.
http://gcc.gnu.org/onlinedocs/gcc-4.3.4/gnat_ugn_unw/
Cited on p. 52.

[2] Bram Adams. *Co-evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems*. PhD thesis, Ghent University, May 2008.
http://hdl.handle.net/1854/11742
Cited on pp. 3 and 102.

[3] Bram Adams. Co-evolution of source code and the build system. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*, pages 461–464, September 2009.
http://hdl.handle.net/10.1109/ICSM.2009.5306272
Cited on p. 102.

[4] Bram Adams, Herman Tromp, Kris De Schutter, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance*, pages 114–123, October 2007.
http://hdl.handle.net/10.1109/ICSM.2007.4362624
Cited on pp. 102 and 107.

[5] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. The evolution of the Linux build system. *Electronic Communications of the ECEASST*, 8:16, February 2008.
http://journal.ub.tu-berlin.de/eceasst/article/view/115
Cited on p. 102.

[6] Christopher Batty. Using the Java Native Interface, 2003. Accessed 2012-02-08.
http://www.cs.umanitoba.ca/~eclipse/8-JNI.pdf
Cited on p. 2.

[7] D.M. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599 – 609, 2003. ISSN 0167-739X.
http://hdl.handle.net/10.1016/S0167-739X(02)00171-1
Cited on p. 20.

[8] Daniel J. Bernstein. Rebuilding target files when source files have changed, 2003.
http://cr.yp.to/redo.html
Cited on p. 8.

[9] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software–-Practice and Experience*, 18(9):807–820, September 1988.
http://hdl.handle.net/10.1002/spe.4380180902
Cited on p. 35.

[10] Ellen Borison. A model of software manufacture. In *Proceedings of an International Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 197–220. Springer, June 1986.
http://hdl.handle.net/10.1007/3-540-17189-4_99
Cited on p. 5.

[11] Ellen Ariel Borison. *Program Changes and the Cost of Selective Recompilation*. PhD thesis, Carnegie Mellon University, July 1989. CMU technical report CMU-CS-89-205. University Microfilms International order number 9023425.
http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-89-205.pdf
Cited on pp. 5 and 83.

[12] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998. ISBN 0471197130.
Cited on p. 75.

[13] Tool Interface Standard Committee. *Executable and Linking Format (ELF) Specification*, 2005. Version 1.2.
http://refspecs.linuxbase.org/elf/elf.pdf
Cited on p. 60.

[14] Tim Daly. Literate programming, 2008. Section 0.4 of volume 4 of axiom source code, 20081101 edition.
https://github.com/daly/axiom/blob/f6fe6f1e553f/books/bookvol4.pamphlet#L2456
Cited on pp. 61 and 68.

[15] Ian J. Davis. *ASX C/C++/Assembler fact extractor*, 2008.
http://www.swag.uwaterloo.ca/asx/
Cited on pp. 98 and 99.

[16] Robert Dewar. The GNAT compilation model. In *Proceedings of the TRI-Ada '94 Conference*, page 58–70, 1994.
http://hdl.handle.net/10.1145/197694.197708
Cited on p. 48.

[17] Paul Duvall. Automation for the people: Remove the smell from your build scripts, 2006.
http://www.ibm.com/developerworks/java/library/j-ap10106/
Cited on p. 104.

[18] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007. ISBN 0321336380.
http://my.safaribooksonline.com/0321336380
Cited on p. 104.

[19] Stuart I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–65, 1979.
http://hdl.handle.net/10.1002/spe.4380090402
Cited on p. 1.

[20] Martin Fowler. Using the Rake build language, August 2005.
http://martinfowler.com/articles/rake.html
Cited on pp. 8 and 70.

[21] Free Software Foundation. Errors in commands, 2002. Section 5.4 of the GNU Make manual, version 3.80.
http://ftp.gnu.org/old-gnu/Manuals/make-3.80/html_node/make_52.html
Cited on p. 90.

[22] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the 1998 IEEE International Conference on Software Maintenance*, pages 190–198, 1998.
http://hdl.handle.net/10.1109/ICSM.1998.738508
Cited on p. 27.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994. ISBN 0201633612.
http://my.safaribooksonline.com/0201633612
Cited on pp. 34 and 75.

[24] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11): 1203–1233, 2000.
http://hdl.handle.net/10.1002/1097-024X(200009)30:11%3c1203::AID-SPE338%3e3.3.CO;2-E
Cited on p. 99.

[25] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.
http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf
Cited on p. 46.

[26] Franco Gasperoni and Patrick Bazire. Smart recompilation and the GNAT compiler. In *Proceedings of the TRI-Ada '94 Conference*, page 104–111, 1994.
http://hdl.handle.net/10.1145/197694.197712
Cited on p. 48.

[27] Daniel M. German. Using software distributions to understand the relationship among free and open source software projects. In *Proceedings of the 2007 IEEE International Workshop on Mining Software Repositories*, 2007.
http://hdl.handle.net/10.1109/MSR.2007.32
Cited on p. 27.

[28] Daniel M. German, Jesús M. González-Barahona, and Gregorio Robles. A model to understand the building and running inter-dependencies of software. In *Proceedings of the 2007 Working Conference on Reverse Engineering*, page 140–149, Oct 2007.
http://hdl.handle.net/10.1109/WCRE.2007.5
Cited on p. 27.

[29] Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. German. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.
http://hdl.handle.net/10.1007/s10664-008-9100-x
Cited on p. 10.

[30] J. H. Griesmer and R. D. Jenks. Scratchpad/1: An interactive facility for symbolic mathematics. In *Proceedings of the 2nd ACM Symposium on Symbolic Algebraic Manipulation*, page 42–58, 1971.
http://hdl.handle.net/10.1145/800204.806266
Cited on p. 58.

[31] Alan Grosskurth. Purely top-down software rebuilding. Master's thesis, University of Waterloo, 2007.
http://hdl.handle.net/10012/2673
Cited on p. 8.

[32] Arnaud Le Hors. *XPM Manual: The X PixMap Format*, 3.4i edition, 1996.
ftp://ftp.x.org/contrib/libraries/xpm-doc-A4.PS.gz
Cited on p. 24.

[33] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010. ISBN 0321601919.
http://my.safaribooksonline.com/0321601919
Cited on pp. 7, 92, and 104.

[34] Ludwig Hähne. Empirical comparison of SCons and GNU Make. Großer Beleg, Technical University Dresden, June 2008.
http://os.inf.tu-dresden.de/papers_ps/haehne-beleg.pdf
Cited on pp. 94 and 105.

[35] Scott Johnston. Interview with Stefan Seefeld of Berlin/Fresco, May 2002.
http://www.advogato.org/article/484.html
Cited on p. 32.

[36] Siim Karus and Harald Gall. A study of language usage evolution in open source software. In *Proceedings of the 2011 IEEE Working Conference on Mining Software Repositories*, page 13–22, 2011.
http://hdl.handle.net/10.1145/1985441.1985447
Cited on pp. 25 and 27.

[37] Steven Knight. Building software with SCons. *Computing in Science & Engineering*, 7 (1):79–88, 2005.
http://hdl.handle.net/10.1109/MCSE.2005.11
Cited on p. 8.

[38] Steven Knight. SCons design and implementation, 2002. Presented at the Python 10 conference.
http://www.python.org/workshops/2002-02/papers/16/
Cited on p. 6.

[39] Phillipe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6): 42–50, November 1995.
http://hdl.handle.net/10.1109/52.469759
Cited on p. 30.

[40] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann, 2000. ISBN 1558604960.
Cited on p. 60.

[41] Camm Maguire. GCL on an AMD64 Linux system? axiom-developer mailing list, 2007.
http://lists.nongnu.org/archive/html/axiom-developer/2007-09/msg00110.html
Cited on pp. 65 and 84.

[42] Jeff Mason and Emil S. Ochotta. The application of object-oriented design techniques to the evolution of the architecture of a large legacy software system. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems*, 1999.
http://static.usenix.org/events/coots99/mason.html
Cited on p. 99.

[43] Steve McConnell. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press, 1996. ISBN 1556159005.
http://my.safaribooksonline.com/9780735634725
Cited on p. 82.

[44] Shane McIntosh. Studying the evolution of build systems. Master's thesis, Queen's University, 2011.
http://hdl.handle.net/1974/6294
Cited on pp. 103 and 107.

[45] Shane McIntosh, Bram Adams, Thanh H. D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 2011 International Conference on Software Engineering*, 2011.
Cited on pp. 97, 103, and 107.

[46] Robert Mecklenburg. *Managing Projects with GNU Make.* O'Reilly, 3rd edition, 2004. ISBN 0596006101.
http://oreilly.com/catalog/make3/book/
Cited on p. 85.

[47] Peter A. Miller. Recursive make considered harmful. *AUUGN: Australian UNIX Systems User Group newsletter*, 19(1):14–25, 1998. ISSN 1035–7521.
http://miller.emu.id.au/pmiller/books/rmch/
Cited on p. 103.

[48] Andrew Neitsch, Kenny Wong, and Michael W. Godfrey. Build system issues in multilanguage software. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, September 2012.
http://andrew.neitsch.ca/build-issues-icsm2012
Cited on p. 4.

[49] Alexander Neundorf. Why the KDE project switched to CMake—and how. *Linux Weekly News*, June 2006.
http://lwn.net/Articles/188693/
Cited on p. 102.

[50] John Ousterhout and John Graham-Cumming. A scalable software build accelerator, 2009. Accessed 2012-08-10.
http://www.electric-cloud.com/downloads/EC-WP_Scalable.pdf
Cited on p. 1.

[51] Enrique Perez-Terron. A rewrite of `tm_axiom.c` (attached). gcl-devel mailing list, 2009.
http://lists.gnu.org/archive/html/gcl-devel/2009-10/msg00000.html
Cited on p. 64.

[52] Dewayne E. Perry, Susan Elliott Sim, and Steve Easterbrook. Case studies for software engineers. In *Proceedings of the 2006 International Conference on Software Engineering*, page 1045–1046, 2006. Slides and handouts available at http://www.cs.toronto.edu/~sme/case-studies/index.html.
http://hdl.handle.net/10.1145/1134285.1134497
Cited on p. 22.

[53] Tim Peters. The Zen of Python, 2004.
http://www.python.org/dev/peps/pep-0020/
Cited on p. 96.

[54] Gregorio Robles. *Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results*. PhD thesis, Universidad Rey Juan Carlos, 2005.
http://libresoft.es/publications/thesis-grex
Cited on pp. 25 and 26.

[55] Jean E. Sammet. Programming languages: History and future. *Communications of the ACM*, 15(7):601–610, 1972.
http://hdl.handle.net/10.1145/361454.361485
Cited on p. 2.

[56] Edmond Schonberg and Bernard Banner. The GNAT project: A GNU-Ada 9X compiler. In *Proceedings of the TRI-Ada '94 Conference*, page 48–57, 1994.
http://hdl.handle.net/10.1145/197694.197706
Cited on p. 45.

[57] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, 1988.
http://hdl.handle.net/10.1145/48022.214505
Cited on p. 83.

[58] Zhong Shao and Andrew W. Appel. Smartest Recompilation. In *Proceedings of the 1993 Symposium on Principles of Programming Languages*, pages 439–450, 1993. ISBN 0-89791-560-7.
http://hdl.handle.net/10.1145/158511.158702
Cited on p. 83.

[59] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 2003 International Conference on Software Engineering*, page 74–83, 2003.
http://hdl.handle.net/10.1109/ICSE.2003.1201189
Cited on p. 9.

[60] Peter Smith. *Software Build Systems: Principles and Experience.* Addison-Wesley, 2011. ISBN 0321717287.
http://my.safaribooksonline.com/0321717287
Cited on pp. 1, 6, 8, 30, 81, 94, 101, and 103.

[61] Joel Spolsky. The law of leaky abstractions, 2002.
http://www.joelonsoftware.com/articles/LeakyAbstractions.html
Cited on p. 104.

[62] Roman Suvorov, Meiyappan Nagappan, Ahmed E. Hassan, Ying Zou, and Bram Adams. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, September 2012.
http://mcis.polymtl.ca/~bram/publications/2012/ICSM2012Roman.pdf
Cited on p. 102.

[63] Nikita Synytskyy, Richard C. Holt, and Ian Davis. Browsing software architectures with LSEdit. In *Proceedings of the 2005 IEEE International Workshop on Program Comprehension*, page 176–178, May 2005.
http://hdl.handle.net/10.1109/WPC.2005.11
Cited on pp. 98 and 99.

[64] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the 2012 International Conference on Software Engineering*, page 650–660, 2012.
http://hdl.handle.net/10.1109/ICSE.2012.6227152
Cited on p. 103.

[65] Walter F. Tichy. Smart Recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986. ISSN 0164-0925.
http://hdl.handle.net/10.1145/5956.5959
Cited on p. 83.

[66] Michael Tiemann. Future of Cygnus solutions: An entrepreneur's account. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution.* O'Reilly, 1999. ISBN 1565925823.
http://oreilly.com/openbook/opensources/book/tiemans.html
Cited on pp. 7 and 89.

[67] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *Proceedings of the 2001 IEEE International Conference on Software Maintenance*, pages 398–407, 2001. Also see the extended version [68].
http://hdl.handle.net/10.1109/ICSM.2001.972753
Cited on pp. 31, 102, and 114.

[68] Qiang Tu, Michael W. Godfrey, and Xinyi Dong. Modelling and extracting the build-time architectural view, 2003. Also see the published version [67].
http://plg.uwaterloo.ca/~migod/papers/2003/btv-ase03.pdf
Cited on pp. 4, 31, and 114.

[69] *JGrok.* University of Waterloo Software Architecture Group, 2008.
http://www.swag.uwaterloo.ca/jgrok/
Cited on p. 99.

[70] William A. Whitaker. Ada—The project: The DoD high order language working group. In *History of Programming Languages II*, page 173–232. Addison-Wesley Professional, 1996. ISBN 0201895021.
http://doi.acm.org/10.1145/234286.1057816
Cited on p. 44.

[71] Adam Wiggins. The twelve-factor app, 2011.
http://www.12factor.net/
Cited on p. 92.

[72] Nathan Willis. On the maintainability of Ruby. *Linux Weekly News*, January 2011.
http://lwn.net/Articles/423732/
Cited on p. 97.

[73] Robert K. Yin. *Case Study Research: Design and Methods.* SAGE Publications, 4th edition, 2009. ISBN 9781412960991.
Cited on pp. 22, 24, 93, 100, and 102.

[74] Erez Zadok. Overhauling Amd for the '00s: A case study of GNU Autotools. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*, June 2002.
https://www.usenix.org/conference/2002-usenix-annual-technical-conference/overhauling-amd-00s-case-study-gnu-autotools
Cited on pp. 7 and 89.