# Build System Issues in Multilanguage Software

Andrew Neitsch, Kenny Wong
*Department of Computing Science*
*University of Alberta*
*Edmonton, Canada*
*Email: {neitsch,kenw}@cs.ualberta.ca*

Michael W. Godfrey
*David R. Cheriton School of Computer Science*
*University of Waterloo*
*Waterloo, Canada*
*Email: migod@uwaterloo.ca*

***Abstract*—Building software from source is often viewed as a "solved problem" by software engineers, as there are many mature, well-known tools and techniques. However, anecdotal evidence suggests that these tools often do not effectively address the complexities of building multilanguage software. To investigate this apparent problem, we have performed a qualitative study on a set of five multilanguage open source software packages. Surprisingly, we found build system problems that prevented us from building many of these packages out-of-the-box. Our key finding is that there are commonalities among build problems that can be systematically addressed. In this paper, we describe the results of this exploratory study, identify a set of common build patterns and anti-patterns, and outline research directions for improving the build process. One such finding is that multilanguage packages avoid certain build problems by supporting compilation-free extension. As well, we find evidence that concerns from the application and implementation domains may "leak" into the build model, with both positive and negative effects on the resulting build systems.**

***Keywords*—build systems, multilanguage software, programming languages**

## I. Introduction

Build systems turn source code into executable programs by orchestrating the execution of compilers, code generators, and other compilation tools. They take as input a specification of which artifacts are produced from which other artifacts by which tools, and perform the required operations to update the project deliverables. A typical example of a build system consists of a Makefile read by Make [1].

Nowadays, build systems may seem to be a solved problem. Java developers using the Eclipse IDE have a "build automatically" option on by default, which provides practically-instantaneous builds with minimal setup. There are other build tools for other languages with similarly advanced convenience, correctness, and speed [2].

But most advances in build tools have focused on building single-language software. What about multilanguage software? The appeal of using multiple programming languages is to exploit their complementary strengths. These benefits, however, may be outweighed by the burden of creating and maintaining a build system that supports multiple languages. For example, Eclipse also automatically builds C/C++ code, and developers commonly integrate Java with C/C++ code via the Java Native Interface (JNI). But Eclipse does not support JNI builds; instead, JNI developers must understand the internals of Java and C++ build tools in order to manually create Makefiles extending Eclipse's auto-generated ones [3, p. 10].

By *multilanguage software* we mean software written in multiple programming languages, in which the parts written in different languages are both necessary and interdependent in the implementation. For example, we would not consider a database server written entirely in C to be multilanguage just because there is a Java client library available, because Tcl is used for running unit tests, or because the documentation viewer is a standalone GUI application written in Python.

Experience with tools such as Eclipse suggests that while there are many mature tools for building single-language software, the situation may be different for multilanguage software. Our investigation is relevant not only to developers and managers maintaining multilanguage projects, but also those considering using new languages on existing projects. Also, this work is applicable to developers and adopters of open-source packages, for whom the inability to build the software in uncertain deployment environments is a serious obstacle to contribution and use. We investigate and summarize the potential issues, and offer research directions.

### A. Overview

We investigate the issues involved in build systems for multilanguage software by examining a selection of open-source multilanguage software packages. Our research questions 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?

As well, we outline potential research implications in terms of improving build processes and tools.

Our qualitative study procedure has three major stages. First, we select five multilanguage software packages from Ubuntu 9.10. Then, we attempt to build the packages and explore both the build problems we encounter and build system features that prevent problems. Finally, we compare observed build problems and means used to avoid those problems across packages, producing both recommendations for practitioners and promising directions of investigation for researchers.

There are three main contributions of this work:

1) In our study, we found that four of the five randomly-selected multilanguage packages have build systems that

978-1-4673-2312-3/12/$31.00 © 2012 IEEE DOI: 10.1109/ICSM.2012.6405265

require manual intervention to build or rebuild a running development version from source code. The fifth also requires intervention in certain cases. We believe that build systems for multilanguage software are, in general, error-prone, and therefore understanding and addressing build system problems is important for software maintenance.

2) We present *patterns and anti-patterns* that summarize the key problems, by comparing how the five different build systems succeed or fail in similar situations.
3) We found that a likely cause of some build problems was the presence of abstractions from application and implementation domains that "leaked" into the build system. This phenomenon can have both positive and negative effects with respect to buildability.

The remainder of this paper gives background information, details our attempts to build each package, describes the build (anti-)patterns, and discusses our results. We conclude with discussions of threats to validity, related work, and a summary.

## II. Background

In this section, we discuss build dependencies, describe the phases of the build process, how they relate to build systems that appear later, and define object-oriented build systems.

The relationships specified among artifacts and tools in a software package are called *dependencies*. They are used for building software from scratch and for incremental rebuilds. After changes to source artifacts of already-built software, it is faster to rebuild only those parts of the software that depend on the changed artifacts. The build system uses dependencies between different artifacts to check whether they are consistent with each other, and brings them up-to-date if not. Dependencies on artifacts that change infrequently, such as compiler versions, can be omitted for faster but less-safe builds. Borison has formalized these concepts [4], [5].

Software builds consist of three main phases: configuration, construction, and packaging. These correspond to finalizing the dependency structure, updating artifacts by traversing the dependency structure, and gathering artifacts for deployment.

- The *configuration* phase entails deciding exactly what to build: which sources, features, compilation tools, libraries, and platform capabilities to use. The output is an automatically buildable system. This phase can involve activities such as scripts determining whether a platform provides support for optional features, or a developer selecting between "Release" and "Debug" variants.
- The *construction* phase uses the configuration to execute the necessary steps to produce end-goal artifacts such as executable binaries, documentation, and unit test reports.
- The *packaging* phase collects the artifacts produced by the build phase for distribution, typically as platform-specific installable packages, e.g., Ubuntu `.deb` files.

There is necessarily some overlap between these phases. Some dynamic configuration decisions may be interleaved with construction, and the final installable package is often an artifact produced by the construction phase.

The complexity of each phase and standard methods for performing them varies by programming language:

- C/C++ software has complicated configuration and construction phases because of the need to support enormous platform variation. The time-consuming nature of C and especially C++ compilation means that C/C++ build systems must correctly support incremental builds.
  autotools [6] is a popular build tool for C/C++ software. It is used by more than 50% of Ubuntu packages with C/C++ source code. It can query for platform details such as word size or available tools and libraries, and/or generate a corresponding Make-based build system. Packaging is often handled by distributing software in source form with an autotools configurator and build system generator.
- Java's write-once-run-anywhere feature results in builds with straightforward configuration and packaging phases. Typically, all source code is byte-compiled to a single JAR archive for distribution.
- The build phase for scripting languages such as Python, Perl, and Ruby involves only optional byte-compilation. Build tools for these languages often provide advanced configuration- and packaging-phase functionality such as installing prerequisite libraries, building C-language extensions, and uploading to a central package repository.

Build tools implemented as APIs in general-purpose programming languages are called *object-oriented build tools*, and build systems dynamically created by such tools are *object-oriented build systems.* Such build systems are composed of objects that can be subclassed, queried, and mutated. This functionality allows projects to dynamically modify their build systems, and to load third-party extension modules that automatically provide build features such as documentation generation, running of unit tests, and production of code quality metric reports. Examples of object-oriented build tools include SCons, Python's distutils, and Ruby's Rake.

Methods developed for specific languages may not be optimal for building multilanguage software. For example, distutils has some support for building C extensions, but that is an artifact-oriented [7] process with configuration-phase emphasis on determining platform capabilities, while Python builds are task-oriented with configuration-phase emphasis on satisfying prerequisites. Using autotools for the Python part of the build, or distutils for the C part, is likely to be problematic due to missing language-specific features, and because the tools are not designed to integrate with each other.

## III. Case Studies

From among the 16 153 packages in Ubuntu 9.10, we selected five multilanguage packages as case studies, without knowing ahead of time how they would build. To assist us in this, we built a tool that used filename patterns similar to those of Robles [8] and Karus and Gall [9] to classify the programming languages of 101 GB of Ubuntu source code. We used these classifications to identify multilanguage packages. The selected packages, shown in Table I, are diverse along several dimensions:

TABLE I
SELECTED MULTILANGUAGE PACKAGES

| Package | Application type | Languages | Industrial | KSLOC | Age (years) | Committers |
|---|---|---|---|---|---|---|
| `synopsis` | Source code documentation | C, Python | No | 94 | 9 | 5 |
| `python3.0` | Programming language | C, Python | No | 585 | 18.5 | 150 |
| `gnat-gps` | IDE | Ada, C, Python | Yes | 450 | 5 | 17* |
| `axiom` | Computer algebra | C, Lisp, Scratchpad | Yes and No | 360 | 37 | 41* |
| `ruby-prof` | Profiler | C, Ruby | No | 4 | 2.5 | 4 |

*Undercounts; the studied `gnat-gps` and `axiom` versions have only 1 and 5 years, respectively, of public commit history

- *Various application types.*
- *Industrial and non-industrial packages.* `gnat-gps` is a proprietarily-developed software product of AdaCore. `axiom` was a closed-source IBM product that was open-sourced after being discontinued as a commercial product. Most of its source was industrially developed, but the current build system was written by open-source volunteers.
- *Various package sizes* as measured in Kilo Source Lines Of Code, i.e., 1 000s of non-blank non-comment LOC.
- *Various ages*, measured in years from the package's first release to the release of the version studied.
- *Various team sizes* and vastly different committers-per-source-line-of-code ratios.

In this early exploratory study, we simply want several multilanguage packages to study, and use random sampling rather than biasing ourselves toward the usual open source subjects, such as Mozilla. The resulting selection is diverse along non-build-system-specific dimensions. For future qualitative studies to investigate along these dimensions in more detail, purposive selection [10] of packages would be appropriate.

We attempt to build each package and consult sources such as documentation, mailing list archives, and source code in order to answer the following questions which address the concerns of user, build expert, and developer stakeholders.

- Overview: What does the software do and which parts are written in which languages?
- Build system: What kind of build system does it use?
- Build problems: What, if any, problems are encountered when building it? When rebuilding it after a change?

In this paper we present major build problems encountered.

For each package, we build the original vendor source code on Ubuntu, to mimic typical build scenarios such as: a Unix end-user installing downloaded source code; a developer making an initial change to a piece of software they use; and a build expert examining a package to improve its build system. We use Ubuntu because it is very popular, and has thousands of packages with plain text metadata for all of them [11], [8].

Our expectation is that each package contains a build system that "just works"—one capable of quickly producing an executable binary without requiring any manual setup, intervention, or source code modification, and that will quickly produce an updated executable after source code changes. We also expect that, to reduce the feedback time from making changes to seeing them working, and to isolate development versions, the build system can skip the packaging phase and the software can run directly from the build directory.

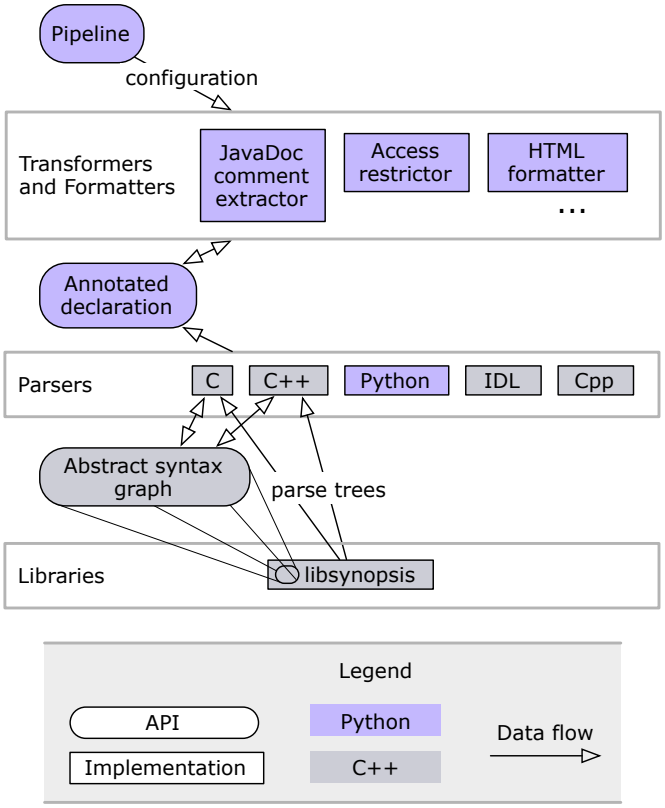


Fig. 1. `synopsis` architecture and languages

One well-known issue with building open-source software is that of missing or incompatible versions of tools and libraries [2, Ch. 15]. One must recursively install compatible versions of all required packages. We attempt to automate this by having Ubuntu's package manager automatically install all build- and runtime-prerequisite packages.

### *Case study 1: `synopsis`*

Overview: `synopsis` is a source-code documentation tool that produces API documentation in formats such as HTML from specially-marked source code comments. It processes Python, IDL (Interface Description Language), C, C++, and unpreprocessed C/C++; these are the languages of an experimental CORBA-based multilanguage windowing package called Berlin/Fresco [12] for which `synopsis` was developed.

The package provides a command-line interface and Python API. It is written mostly in Python. Four of the five source-

code parsers are written in C/C++, but create Python objects internally. The parsers are modified versions of previously-existing open-source packages: the Python parser uses the `compiler` and `tokenize` modules built in to Python, while the IDL, C, C++, and Cpp parsers are based on the third-party omniORB, ctool, OpenC++, and ucpp parsers, respectively.

As an example of a multilanguage architecture, the architecture of `synopsis` is shown in Fig. 1.

Build system: `synopsis` uses the standard Python distutils build tool, and builds the parsers written in C++ with independent, nearly-identical autotools-based builds created by the `synopsis` authors. The builds use the `makedepend` tool to automatically extract dependencies for header files. Because each subproject is built as a shared-library, inter-subproject dependencies between binary objects are not needed.

Build problems: *Multiple library search paths must be manually configured before development builds will run.* The subprojects that are modified versions of independent open-source projects are built independently. Running `synopsis` after it is built is complicated by the fact that the different build systems put their build products into five different temporary folders, all of which must be added to library search paths.

The `synopsis` README file describes a `synopsis`-specific configuration-file option to reduce the number of paths that need configuring. However, the README file notes that builds configured in this way should not be installed, as the installer will "get confused and not install extensions."

*Software does not run on case-insensitive filesystems.* In our first attempt at running the software we built, we were working in Ubuntu using disk space on a Mac OS X network drive. The Mac OS X filesystem is case-insensitive; `synopsis`'s attempts to load the Python `Synopsis.Parsers.Cxx` module collided with the presence of a script called `synopsis.py`. We used a Linux filesystem for the remaining case studies.

*PDF documentation fails to build* due to a bug in the third-party xmlroff tool. xmlroff gives an error about a 'missing namespace binding' for XML input, with an erroneous line number that does not change when the XML input changes. However, the HTML documentation did build successfully.

Summary: `synopsis` is mainly Python with multiple C++ modules that are modified versions of pre-existing open-source projects. The C++ modules are independently built by autotools-based build systems that automatically extract dependencies. Extensive manual adjustment of library search paths is required in order to run. The package has case-collision problems on case-insensitive filesystems. PDF documentation does not build due to a bug in a third-party tool.

*Case study 2: `python3.0`*

Overview: Python is an interactive general-purpose programming language. The interpreter and built-in classes such as `list` are written entirely in C. This separation eliminates bootstrapping issues.

The standard library consists of about 200 top-level modules. About 80% of these modules are written in Python, 30% in C, and 10% using both C and Python.

Build system: The build system uses autotools for configuration. Then a handwritten Makefile compiles together the interpreter and the few standard library modules written in C that are required by distutils. At this point distutils can run, and it builds the rest of the standard library.

Build problems: *We encountered no problems building `python3.0`.* It just worked.

*However, the manually-specified build dependencies can cause rebuild issues.* All interpreter object files are declared as depending on all exported header files; changing any public header file forces a lengthy rebuild of all of Python. But some non-exported header files, such as `Python/importdl.h`, do not appear in any build rules, and changes to them are ignored.

Summary: `python3.0` uses an autotools-based build system with a single handwritten Makefile. This particular handwritten Makefile leads to maintenance problems. There are no bootstrapping issues in running Python before Python is built, because the language core is implemented entirely in C.

*Case study 3: `gnat-gps`*

Overview: `gnat-gps`, or GNAT Programming Studio, is an IDE primarily for Ada but also C/C++, developed by AdaCore, who also develop the GNAT Ada compiler.

Much of `gnat-gps` is written in Ada, with significant portions of the user interface implemented using a Python plugin mechanism. `gnat-gps` uses the GNAT Ada compiler, also written in Ada, as a linked library for Ada parsing. Nearly all of `gnat-gps`'s C code is a modified version of Red Hat's Source-Navigator C/C++ fact extraction and indexing package. It executes in a separate process and we do not consider it part of `gnat-gps` proper.

The mechanism for extending the user interface via Python plug-ins is also available to end-users who save Python scripts in `~/.gps/plug-ins`. It provides concise access to the IDE's object model. For example, there is a 60-line example plug-in that implements a soft tabs feature, so that pressing the tab key in the source editor inserts the syntactically appropriate amount of indentation rather a literal tab character.

Build system: The build system uses the GNAT Project Manager, or `gpr` for short. It is an Ada-specific build tool that implements hierarchical build systems for Ada [13]. Project files for `gpr` are text files with Ada-like syntax.

There is also a hierarchy of Makefiles to build C portions of the source code, with a structure similar to that of the Ada build hierarchy.

Build problems: We encountered many problems trying to build `gnat-gps`; ultimately, we were unable to build using the vendor-supplied build system, and were forced to use a replacement written in frustration by the Ubuntu maintainer.

*`gpr` failed to parse conditionals in the project files*, even though they were nearly identical to that in the manual. We had to manually remove each conditional.

*Link errors due to incompatible changes in Gtk* prevented us from building. There were struct field name differences such as `GtkEntry.x_n_bytes` versus `GtkEntry.n_bytes` between the system GUI toolkit library and its Ada bindings.

*Ubuntu packager rewrites build system.* Others have also struggled with this build system. We eventually examined the Ubuntu patches for `gnat-gps` and found that the Ubuntu packager for `gnat-gps` had written a new build system. In the source code of his replacement `gps.gpr`, he complains about the vendor system's "complex structure of project files importing and including each other, brittle configure scripts, [and] evil recursive Makefiles and Makefile fragments."

He replaced dozens of `gpr` project files and Makefiles with one `gpr` file and one Makefile. All non-Source-Navigator C code is compiled into a static library, all Ada code is compiled as a single project, then the two are linked together.

Once properly configured to find all of the source code, `gpr` handles all Ada source file dependency issues automatically.

*Once built, the software did not run properly from the build directory.* Until it was installed, icons were missing, and many dialogs which are rendered from XML templates were blank.

Summary: `gnat-gps` is difficult enough to build that the Ubuntu maintainer rewrote the build system. However, most of these build system problems are not an issue for end-users because the Python plug-in mechanism requires no building.

*Case study 4: `axiom`*

Overview: `axiom` is an interactive computer algebra package that implements a language called Scratchpad. It was originally developed in 1971 for IBM mainframes [14]. It includes a library of definitions and algorithms for hundreds of algebraic structures in mathematics and computer science.

Most of the source code is in literate format, consisting of 1.5 million lines of mixed source and documentation expressed in LaTeX macro code. Extracted to individual source files, it is roughly 360 KSLOC. The current maintainer, once a developer of `axiom` at IBM, chose to convert to a literate format to preserve knowledge of how and why the package works.

The Scratchpad interpreter-compiler is written in Lisp and compiles Scratchpad to Lisp, which can then be interpreted or compiled to C, which can be compiled to machine code. All of the mathematical code is written in Scratchpad. `axiom`'s interface is primarily a teletype, but there are rudimentary graphics and documentation interfaces implemented in C that communicate with `axiom` via sockets.

Build system: The source code is embedded in LaTeX files individually as large as 7 MB. A provided tool extracts individual source files and Makefiles from the LaTeX code and calls Make. All changes must be applied to the LaTeX sources, which must then be extracted *en masse* and recompiled from scratch. There is no provision for incremental builds.

The major steps of the build process are:

1) Build GCL (GNU Common Lisp) and statically link `axiom`-specific routines into the Lisp binary.
2) Build the Scratchpad interpreter-compiler.
3) Compile the math libraries.
4) Run regression tests.

Build problems: *We were unable to build `axiom` at all on Ubuntu 9.10 x86_64.* `axiom`'s build system was so problematic that we consider it an outlier and exclude its more unusual build problems from consideration of general build-system problems. We experienced many build problems with GCL and then `axiom`, working through mysterious errors such as `Command not found` and `cannot trap sbrk` and `Cannot read address` and `I'm not an object` before narrowing down the problem.

*`axiom` relies on GCL, which is broken on this version of Ubuntu.* The fundamental issue preventing the build from succeeding is that GCL's `compiler::link` function does not work on this version of Ubuntu, due to incompatibilities between system tools and GCL's nonstandard symbol tables, which are appended to the end of the binary object files outside any section structure. This is not the first time `axiom` has been unbuildable; the maintainer documents in section 0.3.2 of the source code that when `axiom` was open-sourced, it had been ported from GCL to Codemist Common Lisp (CCL), but he ported it back to GCL, being unable to get the CCL-based version to build. However, a build on Ubuntu 10.10 using the Ubuntu build script did work on the first try.

*The build system inadvertently ignores build errors.* Consider this Makefile fragment:

```
gcldir: ; cd ${GCLVERSION}; \
        ./configure ${GCLOPTS}; \
        ${ENV} ${MAKE}; \
        echo '(progn (load "lsp/sys-proclaim.lisp")' \
        '(system::save-system "${OUT}/lisp"))' \
        | unixport/saved_gcl
```

It attempts to configure GCL, build GCL, and save a new lisp image, without stopping if any of the commands fail. These shell commands need to be joined with conditionals '`&&`' rather than the '`;`' sequential list separator.

*Summary* `axiom` is impossible to build on Ubuntu 9.10 x86_64 because it relies on a broken lisp, but other aspects of the build system such as lack of error detection and absence of incremental builds make it difficult to determine even that.

*Case study 5: `ruby-prof`*

Overview: `ruby-prof` is a profiler for the Ruby programming language. It can be used as a command-line tool or as a library, providing raw profile data output or formatted cross-referenced HTML output. The profile data capture module is written in C, and everything else is written in Ruby.

Build system: The `ruby-prof` project uses Rake, an object-oriented build system implemented in Ruby. Strangely, the C module is not built by the Rake build system. Rake handles unit tests and packaging, but instead of building the C module, Rake records the need for a build in metadata processed at installation time. At installation time, Makefiles are generated and the C profile capture module is built.

`ruby-prof` provides a `RubyProf::ProfileTask` class to integrate profiling into the builds of other Rake projects.

Build problems: *The `mkmf`-library-generated Makefile for the C extension has no dependency rules on header files.* Make will unhelpfully say, "Nothing to be done" if declarations in header files have changed but the `.c` file has not.

*There is no mechanism to keep the source code and a platform-specific binary in synchronization.* Having compila-

TABLE II
BUILD ISSUES

| Patterns and Anti-patterns | synopsis | python3.0 | gnat-gps | axiom | ruby-prof |
|---|---|---|---|---|---|
| Anti-pattern: Filename Collision | - case collision | | | | |
| Anti-pattern: Installation Required | - requires excessive runtime path setup | + automatically checks if in build directory | - fails to load resources from build directory | | |
| Anti-pattern: Unverified Third Party Software | - uses buggy xmlroff | | - gpr failure<br>- incompatible Gtk header file change | - requires GCL, which fails to work | |
| Anti-pattern: Ignored Error | + build error detected immediately | | + build error detected immediately | - GCL build failures ignored | |
| Anti-pattern: Incorrect Dependencies | + automatic dependency extraction | - manually specified dependencies are incorrect | + gpr handles dependencies automatically | | - auto-generated build system omits all dependencies |
| Pattern: Build-Free Extensibility | | + can extend with scripts | + python plugin mechanism | + can extend using Scratchpad language | |
| Pattern: Object-Oriented Builds | | + distutils used for both standard library and third-party extensions | | | + package exports build functionality<br>+ library upgrades can fix build problems |

tion not occur until installation time can be problematic. Windows machines do not usually have compilers. ruby-prof addresses this with a pre-built Windows .dll module included in the source distribution. But there is no automated mechanism to keep it up-to-date with the source code.

*Summary* ruby-prof is tiny compared to other projects in this selection. But there are still build system issues. However, these are arguably bugs in standard Ruby libraries like mkmf that could improve as the ecosystem matures. For example, recent releases of ruby-prof omit pre-built Windows .dlls, because newer versions of the Windows Ruby installer link to a package containing a C compiler to allow C extensions and Ruby packages to be built at install time as intended.

## IV. BUILD PATTERNS AND ANTI-PATTERNS

By comparing build problems across the five different packages, we see both common build problems among the packages and how other packages avoid the same problems. We summarize these findings in terms of build *patterns* and *anti-patterns* [15], which cover the major issues encountered.

We use the following template to structure the findings:

- *Description*: What is the pattern or anti-pattern?
- *Consequences*: What are its potential outcomes?
- *Evidence*: What evidence is there that it exists, particularly in the case studies?
- *Remedies*: For an anti-pattern, how can it be fixed?
- *Applicability*: Does it apply to software implementation, build systems, or builds of multilanguage software?
- *Research directions*: What research work could help address, support, or explore this pattern or anti-pattern?

The distinction between *remedies* and *research directions* is that remedies can be implemented in the short-term by practitioners, while research directions represent more systematic studies, solutions, and tools to produce in the longer term.

The *applicability* heading also helps to identify which (anti-)patterns may apply to single-language software.

The findings are summarized in Table II, where a minus sign indicates build issues that are instances of an anti-pattern, and a plus sign indicates build issues that exemplify a pattern or resolve an anti-pattern.

### *Anti-pattern: Filename Collision*

*Description:* Source code files have names that cause build or run problems on certain filesystems. Similarly-named files may be confused on case-insensitive filesystems. Specifications of legal filenames may conflict across filesystems.

*Consequences:* Software may not build or run on some operating systems, or on clients of certain network filesystems.

*Evidence:* When running synopsis on Ubuntu using a filesystem served by Mac OS X, the Synopsis module cannot be loaded since it is confused with the synopsis.py script.

*Remedies:* Files must be renamed to avoid conflicts. Meaningful replacement names likely require human input.

*Applicability:* This anti-pattern is a build-system-specific issue. There may be cross-language filename compatibility issues, but we do not encounter any in the study.

*Research directions:* Projects that support a variety of platforms are likely to have discovered and resolved this issue. In general, however, it would be useful to investigate and devise lint-like checks for the portability of build systems.

### *Anti-pattern: Installation Required*

*Description:* A newly-built software package must be installed before it can be run.

*Consequences:* New developers may believe they made a mistake when their newly-built software does not run. For all developers, it is slow to re-install a package after every compilation. Care is also needed to avoid installations of development software from disrupting non-development activity.

*Evidence:* synopsis requires extensive manual path setup to run from its build directory. gnat-gps also has problems running from its build directory, such as icons and certain dialogs not showing up properly. However, python3.0 has

startup routines that check whether it is running from its build directory, and adjusts search paths accordingly.

*Remedies:* One way to address this problem for small projects is to keep the file layout of the build as close to that of the deployment as possible. For larger projects, specific remedies such as `python3.0`'s startup check may be needed.

*Applicability:* This is a general build issue, but is more of an issue in multilanguage software because search paths and resource loading must be set correctly for multiple languages.

*Research directions:* Exploration of the underlying path management issue could lead to standard libraries for managing resource paths, including between languages. Another potential improvement is devising general support for the notion of "run where built" in build frameworks.

*Anti-pattern: Unverified Third Party Software*

*Description:* A package relies on third-party software, but this software is used without specifically testing at configuration time whether the required functionality works.

*Consequences:* A bug or incompatibility in third-party software may cause the build to fail in a way that makes the original package seem at fault.

*Evidence:* `synopsis` was unable to produce PDF documentation due to a bug in xmlroff. The Ada build tool `gpr` was unable to handle certain syntax in `gnat-gps`'s build scripts. Despite `axiom` shipping code and patches for three versions of GCL, `axiom` could not be built at all due to an incompatibility between the GCL Lisp implementation and the Ubuntu Linux object file format. These problems occurred despite using the Ubuntu package manager to install all the prerequisites.

*Remedies:* Consider how autotools checks that the C compiler works as expected. That is, autotools tries to compile a small C program and verify that the compiler produces a runnable executable with the expected behaviour. If this fails, it may try to find and use other C compilers. Otherwise, the build is aborted with a notice saying to fix the C compiler before the build can proceed. Similar tests can be done at configuration time for other third-party software.

*Applicability:* This is relevant to all build systems, but may be more common in build systems for multilanguage software due to use of less-popular tools such as `gpr` and GCL for less-popular languages such as Ada and Lisp. GCL in particular has not had an official release on its home page since 2005.

*Research directions:* Maintaining compatibility between parallel evolving software projects is a general software problem. However, in the realm of builds, one could expand the autotools approach by gathering and developing basic functionality and compliance tests for other common software, and allow these tests to be used by a variety of different build frameworks.

*Anti-pattern: Ignored Error*

*Description:* Errors in the build process are not detected or trapped immediately.

*Consequences:* The root cause of a failed build may have occurred much earlier than when the failure is manifested, making it more difficult to pinpoint.

*Evidence:* `axiom`'s build system tries to compile GCL but inadvertently continues even if the compile fails, and the result is an error later in the build when GCL is needed. For `synopsis`, `python3.0`, and `gnat-gps`, while the build errors we saw led to immediate halts, their Makefiles have build steps whose failure could be ignored inadvertently.

This anti-pattern is common when using Make with complex inline shell commands. While Make will abort a build if an individual shell command returns an error, by default it will abort sequences of commands chained by the `;` operator or in `for` loops only if the *last* command fails. This subtle distinction is not well-known. One popular Make textbook briefly mentions it as "one of the most important aspects of programming robust Makefiles" [16, p. 236], but many examples throughout the same text exhibit this anti-pattern.

*Remedies:* Developers can detect more errors in Make by using `&&` instead of `;`, or by adding `SHELL = /bin/sh -e` at the beginning of Makefiles. However, this involves understanding the subtle differences in error-handling behaviour of various shell script operators.

Other specific fixes may be needed for other build tools.

*Applicability:* Ignoring error codes is a general software issue. However, in the realm of build systems, the interacting semantics of Make and shell scripting make it easier for errors to be ignored inadvertently.

*Research directions:* The specific issue with Make is largely amenable to automated detection and correction. More generally, ignored build errors could be detected by tracing tools that monitor the return status of all executed commands to abort the build if a critical one fails. However, developers will sometimes need to intentionally ignore errors from some commands.

*Anti-pattern: Incorrect Dependencies*

*Description:* Build dependencies are specified manually, which is error-prone.

*Consequences:* Time is wasted rebuilding parts that do not need to be rebuilt, while parts that need to be rebuilt are not.

*Evidence:* For its C/C++ parts, `synopsis` uses autotools-generated build systems that automatically extract dependencies, and that even work between projects due to the use of shared libraries. The Ada build tool used in `gnat-gps` handles all source dependency issues automatically [13]. `python3.0` uses manually-specified dependencies that are incorrect—many header file changes forces full rebuilds, while some are ignored by the build system. `ruby-prof` uses an automatically-generated build system that fails to include dependencies for any of its nine C header files.

*Remedies:* Use a build tool that generates or resolves build dependencies automatically.

*Applicability:* This is a build-system-specific issue. Because so much build system research focuses on dependencies, we expected that looking at multilanguage packages would turn up interesting cross-language dependency issues, but that is not the case. This may be merely due to the small sample size.

*Research directions:* Most prior research has focused on this concern. Surprisingly, it was not a major issue in our study; even the `python3.0` rebuild issue is somewhat contrived.

*Pattern: Build-Free Extensibility*

*Description:* Contributions to projects are incorporated through a run-time extensibility mechanism.

*Consequences:* Build system problems are avoided by using extensibility mechanisms that do not require building the main software package.

*Evidence:* All case studies have a dynamic extension mechanism. This may be a common design in multilanguage software. In particular, `python3.0`'s library can be extended by writing Python source code files. `gnat-gps` can be extended with a Python-language plugin mechanism that only requires placing a source code file in a special directory. `axiom`'s mathematical capabilities can be extended using the Scratchpad language it implements. Also, `synopsis` and `ruby-prof` provide object-oriented extension mechanisms.

*Applicability:* This is not a multilanguage issue *per se*, but adding build-free extensibility mechanisms is usually implemented via adding a scripting interface. Once that is done, developers can find it easier to make some types of core changes via the scripting interface—as happened with `gnat-gps`—and the package becomes multilanguage.

*Research directions:* Empirical studies could verify our speculation about whether projects with build-free extension mechanisms have more active communities, and whether adding such mechanisms is an evolutionary force towards becoming a multilanguage package.

*Pattern: Object-Oriented Builds*

*Description:* The build system can be dynamically customized and extended by project developers and by third-party build libraries.

*Consequences:* Object-oriented build systems bring benefits such as encapsulation and reuse into build systems. Problems in a common build library can be fixed in one place and propagated wherever it is used, potentially across many projects.

*Evidence:* `ruby-prof`'s build system problems are largely encapsulated inside libraries that are upgraded by third parties. `ruby-prof`'s build system also can export its functionality, namely profiling, to the build systems of other projects. `python3.0`'s object-oriented distutils build system is used for its standard library but also externally as a well-supported tool for building Python-language projects such as `synopsis`. Problems we did see in `python3.0`'s build system were with the custom non-distutils portion.

*Applicability:* This pattern is build-system-specific. The additional use of encapsulation may help to address the increased complexity of multilanguage builds.

*Research directions:* Standardized interfaces for tools such as compilers and profilers would allow greater experimentation in build system design. Encapsulation of build system processes for various languages could also be used for more convenient build systems for multilanguage software.

## V. Discussion

### A. Using build patterns and anti-patterns

With respect to RQ1, the build patterns and anti-patterns help to summarize and organize the key issues identified in the studied multilanguage packages. From the case studies, we believe the developers were unaware of certain build problems, at least within their own environment. For RQ2, the patterns embody suggested designs for the build process and the anti-patterns offer remedies to common build problems.

Nevertheless, fashioning a scalable build system or addressing build problems does take effort to implement. A particular project may have other higher priorities, and may be fine in terms of the technical debt in its build system. In practice, build system improvements need to be selective.

Anti-pattern *Incorrect Dependencies* can be addressed systematically through automated tools. Other anti-patterns need researchers to build tools, but presently, *Filename Collision* and *Unverified Third-Party Software* could be addressed manually on a case-by-case basis. For example, rather than writing tests for all third-party packages, specific tests for key features or known buggy packages would be advised. Some anti-patterns may require substantial work: a thorough line-by-line analysis of the build system for *Ignored Error*, and potential implementation rearchitecting for *Installation Required*.

An interesting question is whether popular packages are correlated with better build systems. Popular packages may receive and fix more build system bug reports. It may be that packages that are easy to build can lead to a vibrant community. Or perhaps popular packages grow to involve build experts who can design effective build systems. This correlation needs further research.

### B. Abstraction

Interestingly, we find that the abstractions, mental models, business goals, and design philosophies used in the application and implementation domain may "leak" into the build system space. This phenomenon may be a natural consequence due to familiarity, available skills, time pressures, or the domains, but can have inconsistent, *i.e.*, positive or negative, effects on buildability. This may provide a partial answer to research question RQ3. Consider the five case studies:

- `synopsis` integrates multiple independently-developed and independently-built software components using a standardized interface. The approach works well, except for some issues with setting up paths for the different parts to be able to load each other.
  This approach is similar to those of the experimental CORBA-based windowing package that `synopsis` was developed to support. Service lookup and path configuration are also common issues in getting CORBA applications to run.
- `python3.0`'s build system shares many qualities with idiomatic Python code, such as being simple, flat, and explicit [17]. The result is a small and efficient build system. However, the desire to be explicit and avoid

so-called 'magic' such as dependency extractors is also the cause of `python3.0`'s rebuild issues, where explicit lists of incorrectly-specified build dependencies lead to lengthy and sometimes erroneous rebuilds.

- `gnat-gps` is shipped with a complicated hierarchical build system that the Ubuntu maintainer needed to rewrite. However, the hierarchical build system is a natural outcome of the business goals of AdaCore, the corporate developer of `gnat-gps`. They intend components of `gnat-gps` to be split off as independent open-source components, making Ada more attractive for development and increasing sales of commercial Ada offerings.
- `axiom` provides a real-time interactive "Scratchpad" for doing mathematics. In contrast, changing `axiom`'s internals is a complicated process of marking up code in LaTeX, rebuilding the software from scratch, and running comprehensive regression tests to ensure correctness. However, this is similar to procedures used by some professional mathematicians, who may initially develop ideas on scratch paper, then LaTeX their results and submit them to a lengthy review process prior to publication.
- `ruby-prof`'s build system is adequate and what issues do exist are likely to be resolved by future upgrades to the build libraries it uses. This mirrors the object-oriented developer-focused Ruby philosophy [18], where frequent change is acceptable, experimentation is valued, and bugs are fixed by upgrades.

While this finding is speculative, it suggests that the applicability of mental models and abstractions in the application and implementation domains can affect build system quality. In terms of research, this issue could be further explored through systematic studies of the different concepts that people use to think and communicate within the application, implementation, and build domains.

One implication could also be that a successful build requires an independent build expert that is less tainted by the peculiarities of a specific application or its implementation technology. Indeed, McIntosh et al. conjecture that concentrated build ownership results in less build maintenance effort than dispersed build ownership for open source packages [19].

## VI. Threats to Validity

In this section we address potential sources of bias using the standard criteria [20].

*Construct validity: Were we actually measuring what we sought out to measure?* We set out to determine the build issues of multilanguage software. One possible source of bias is our use of Ubuntu, both as a source of packages and platform on which to do our builds. Ubuntu is not necessarily supported directly by any of the packages. However, Ubuntu is extremely popular and a large portion all people building these packages would do so on Ubuntu, so it is reasonable to use it as a build platform. Additionally, we used the original vendor source code for the packages, and none of the issues that we experienced were Ubuntu-specific, except perhaps the incompatibility with GCL when building `axiom`.

*Internal validity: Are our casual inferences the result of coincidence or unknown third factors?* There are no serious threats to internal validity within the main body of this exploratory study, which are about the behaviour of build tools on given inputs. Internal validity does apply to where we attempt to explain *why* build problems happen, in Subsection V-B, which is admittedly speculative.

*External validity: How generalizable are the findings?* We only looked at open-source Unix packages, and while they are from different application areas, they are all programming language implementations and tools. Since they were randomly selected, their similarity may be a result of coincidence, a property of Ubuntu, open-source, or multilanguage software ecosystems, or bias in our language identification tool. The issues we encountered may or may not be widespread, but they are still real, and come from real software.

*Reliability: Is the result dependent on the researchers or tools?* Only one investigator examined the software packages. The initial setup, selection of packages, and procedure to get to the first build system error message are all completely reproducible. After this point, however, the measures taken and subsequent build problems encountered could vary widely for different investigators. The influence of experience and preconceived opinions about the packages is addressed somewhat through the random selection. The only selected package that the investigator had heard about prior to this study was `python3.0`. Additional investigators could perform the builds and compare notes, but involving several people to duplicate deep analyses of build problems seems too heavyweight for an exploratory study.

The conclusions and (anti-)patterns inferred could also vary between investigators. We have presented some of the more prevalent and/or interesting (anti-)patterns that we encountered in this study, but there certainly are other ones that could be inferred and that other investigators would focus on instead.

## VII. Related Work

Adams developed visualization and refactoring tools for Make-based build systems [21] that integrated aspect-oriented programming, a multilanguage technique, into legacy C software. His case studies contained deep analyses of Make build systems including Quake and the Linux kernel [22]. We start with multilanguage software using a variety of build tools in addition to Make.

Smith's practitioner-oriented book about build systems [2] has an extensive section on "Various Ways to Reduce Complexity." Some of his suggestions, such as "Use a Modern Build Tool," "Automatically Detect Dependencies," and "Abort the Build After the First Error" are consistent with our findings. We address multilanguage software and structure our findings in the form of patterns and anti-patterns.

Tu and Godfrey conducted case studies of build-time software architectures, such as the techniques that compilers use to compile themselves, or that Perl, mainly written in Perl, uses to compile itself [23]. They focus on the build-time software

architecture view, while we focus on issues in getting software to build at all.

Tamrawi et al. developed a static analysis tool that can detect certain "code smells" in Makefiles [24]. We study the behaviour of several different build systems, not only Make.

Hähne empirically studied the relative scalability and performance of make and SCons using a synthetic project generator [25]. We discuss the buildability of real projects.

Several practitioner-oriented books on continuous testing and deployment discuss creating fully-automated builds within an organization [7], [26]. A co-author of one such book has written a developer article [27] describing "build smells," which are similar to our build patterns and anti-patterns.

## VIII. Conclusion

Build systems for software are often viewed as a solved problem, particularly for single-language software. We qualitatively investigated build issues from a selection of five multi-language software packages. Surprisingly, we found significant problems in getting the software to build at all.

Our key finding is that many build problems can be systematically addressed. Relatively less focus in research has been placed on the build quality or "buildability" of software, versus other software qualities. Yet, this quality is an important aspect for growing a developer community around an open source software project. Consequently, understanding and addressing build problems is an important activity in maintaining software, which needs better support and further research. We summarize our findings by describing a preliminary set of build patterns and anti-patterns. As described in each (anti-)pattern's *applicability* heading, many (anti-)patterns also apply to building single-language software.

As future work, we plan to explore tool support for understanding multilanguage packages and their complexities, specifically to address build anti-patterns. By exploring more build systems, we wish to expand the set of build patterns and anti-patterns. Further study is needed on unique comprehension issues in build systems, particularly the phenomenon of mental models "leaking" into the build system.

## Acknowledgements

We thank Abram Hindle for reviewing a draft of this paper. We also thank the anonymous reviewers for their comments and suggestions. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

## References

[1] S. I. Feldman, "Make—A program for maintaining computer programs," *Software—Practice and Experience*, vol. 9, no. 4, pp. 255–65, Apr. 1979. http://hdl.handle.net/10.1002/spe.4380090402

[2] P. Smith, *Software Build Systems: Principles and Experience*. Addison-Wesley, 2011, ISBN 0321717287. http://my.safaribooksonline.com/0321717287

[3] C. Batty, *Using the Java Native Interface*, 2003, accessed 2012-02-08 [Online]. Available: http://www.cs.umanitoba.ca/~eclipse/8-JNI.pdf.

[4] E. Borison, "A model of software manufacture," in *Proc. Int. Workshop Advanced Programming Environments*, ser. Lecture Notes in Computer Science, vol. 244. Springer, June 1986, pp. 197–220. http://hdl.handle.net/10.1007/3-540-17189-4_99

[5] E. A. Borison, "Program Changes and the Cost of Selective Recompilation," Ph.D. dissertation, Carnegie Mellon University, Jul. 1989, University Microfilms International order #9023425. http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-89-205.pdf

[6] E. Zadok, "Overhauling Amd for the '00s: A case study of GNU Autotools," in *Proc. FREENIX Track USENIX Annual Tech. Conf.*, Jun. 2002. https://www.usenix.org/conference/2002-usenix-annual-technical-conference/overhauling-amd-00s-case-study-gnu-autotools

[7] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, ISBN 0321601919. http://my.safaribooksonline.com/0321670256

[8] G. Robles, "Empirical Software Engineering Research on Libre Software: Data Sources, Methodologies and Results," Ph.D. dissertation, Universidad Rey Juan Carlos, 2005. http://libresoft.es/publications/thesis-grex

[9] S. Karus and H. Gall, "A study of language usage evolution in open source software," in *Proc. 8th Working. Conf. Mining Softw. Repositories*, 2011, pp. 13–22. http://hdl.handle.net/10.1145/1985441.1985447

[10] D. E. Perry, S. E. Sim, and S. Easterbrook, "Case studies for software engineers," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 1045–1046, slides and handouts available at http://www.cs.toronto.edu/~sme/case-studies/index.html. http://hdl.handle.net/10.1145/1134285.1134497

[11] J. M. Gonzàlez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German, "Macro-level software evolution: A case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, 2009. http://hdl.handle.net/10.1007/s10664-008-9100-x

[12] S. Johnston, "Interview with Stefan Seefeld of Berlin/Fresco," May 2002. http://www.advogato.org/article/484.html

[13] R. Dewar, "The GNAT compilation model," in *Proc. Conf. Tri-Ada '94*, 1994, pp. 58–70. http://hdl.handle.net/10.1145/197694.197708

[14] J. H. Griesmer and R. D. Jenks, "SCRATCHPAD/1: An interactive facility for symbolic mathematics," in *Proc. 2nd ACM Symp. Symbolic Algebraic Manipulation*, 1971, pp. 42–58. http://hdl.handle.net/10.1145/800204.806266

[15] W. H. Brown, R. C. Malveau, H. W. S. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998, ISBN 0471197130.

[16] R. Mecklenburg, *Managing Projects with GNU Make*, 3rd ed. O'Reilly, 2004, ISBN 0596006101. http://oreilly.com/catalog/make3/book/

[17] T. Peters, "The zen of Python," 2004. http://www.python.org/dev/peps/pep-0020/

[18] N. Willis, "On the maintainability of Ruby," *Linux Weekly News*, January 2011. http://lwn.net/Articles/423732

[19] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011. http://hdl.handle.net/10.1145/1985793.1985813

[20] R. K. Yin, *Case Study Research: Design and Methods*, 4th ed. SAGE Publications, 2009, ISBN 9781412960991.

[21] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Proc. IEEE Int. Conf. Softw. Maint.*, October 2007, pp. 114–123. http://hdl.handle.net/10.1109/ICSM.2007.4362624

[22] B. Adams, "Co-evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems," Ph.D. dissertation, Ghent University, May 2008, ISBN 9789085782032. http://hdl.handle.net/1854/11742

[23] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proc. IEEE Int. Conf. Softw. Maint.*, 2001, pp. 398–407. http://hdl.handle.net/10.1109/ICSM.2001.972753

[24] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 650–660. http://hdl.handle.net/10.1109/ICSE.2012.6227152

[25] L. 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

[26] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007, ISBN 0321336380. http://my.safaribooksonline.com/0321336380

[27] P. Duvall, "Automation for the people: Remove the smell from your build scripts," 2006. http://www.ibm.com/developerworks/java/library/j-ap10106/