How We Manage Portability and Configuration with the C Preprocessor

Andrew Sutton and Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{asutton, jmaletic}@cs.kent.edu

Abstract

An in-depth investigation of C preprocessor usage for portability and configuration management is presented. Three heavily-ported and widely used C++ libraries are examined. A core set of header files responsible for configuration management is identified in each system. Then macro usage is extracted and analyzed both manually and with the help of program analysis tools. The configuration structure of each library is discussed in details and commonalities between the systems, including conventions and patterns are discussed. A common configuration architecture for managing portability concerns is derived and presented.

1. Introduction

In a seminal paper on software aging, Parnas describes two distinct reasons why software ages: 1) failure to meet changing needs and 2) evolution at the hands of its own developers [15]. Parnas rightly lays the blame for aging at the hands of consumer demand and developmental difficulties, but seems to ignore the impact of the evolution of platforms for which the software was originally developed. Ironically, he claims that software written 40 years ago would function perfectly today, if he could just find a computer that was phased out of existence 30 years ago. Did this software age because today's users would expect more, or did it fail because it did not evolve along with its computer platform?

It seems that software aging can also be blamed on the inability (or indifference) of developers to adapt to evolving or different platforms. However, this could also be seen as an indirect influence of consumer demand on the software – the demand for better computers, operating systems, and programming languages all impact the need to adapt existing software. For example, with the release of Vista, Microsoft has deprecated a number of API's, and as an unfortunate result, some programs may crash or even fail to run at all. Because of these shifting foundations, organizations often release different configurations of their software targeting alternate operating systems and/or versions thereof. From a maintenance perspective, we understand this as software *portability*.

Although we often think of portable software as that which targets different operating systems such as Linux and Windows, the differences need not be so great. Even small differences in OS platforms can result in widely variant behavior in the software. Moreover, differences can occur at every level of software – from the implementation and semantics of system calls to the name lookup mechanism in a compiler.

Developing and maintaining portable software is costly because each new variant or version of a dependency increases the number of build configurations and therefore increases maintenance and testing efforts. Newly introduced configurations require integration and testing. Legacy configurations may be retained or eliminated; if retained, they will require general maintenance. Without a general strategy and reasonable software architecture for handling this multiplicity of configurations, their addition/removal can wreak havoc on the structure of the source code.

In C and C++ programs portability is invariably managed using the C preprocessor. Source code is conditionally compiled depending on variables of the compiler, operating system interface, and other dependent libraries. In this paper, we are interested in determining how popular systems achieve portability with the C preprocessor. To this end, we studied the preprocessor-based configuration of three widely used and heavily ported software libraries: the Qt GUI Toolkit, the Adaptive Communications Environment (ACE), and the Boost C++ Libraries. Specifically, we developed tools to help us extract data from the preprocessor directives. We then analyze this data in order to discern emergent patterns for managing portability.

As a result of this study, we discovered a number of techniques for building portable software that are common between the three systems. These techniques are used to build abstractions out of preprocessor macros, enabling developers to separate dependency concerns.

Most importantly, we found that all three share a common preprocessor-based, software architecture that provides an extensible foundation for future adaptation to different compilers, operating systems, and versions thereof.

The paper is organized as follows. In section 2, we discuss the approach and tools used to study the different libraries. Section 3 contains a detailed description of the preprocessor-based configurations of the systems. In section 4, we discuss emergent techniques and patterns for portability and configuration. Lastly, the common configuration architecture is described in section 5.

2. Methodology

Our study of the different libraries focused on two primary elements of the C preprocessor: 1) include files and 2) configurable macros. To facilitate this work, we abstracted the familiar notions of file inclusion and macro definition (and evaluation) to represent a more programmer-centric model of the concepts. Specifically, we define *include files* as the unique names of files referenced in include directives. This allows us to associate each inclusions with two distinct files, thereby simplifying include graph extraction.

For macro analysis, we defined a *configurable* macro in a similar manner. Specifically, these are the unique names of symbols appearing in definition, undefinition, and condition directives. This model of macros allows us to associate all known (if any!) values with each identifier, and all distinct usages in preprocessor conditions.

We implemented two tools to extract and model these concepts. Both tools leverage our srcML [3, 13] infrastructure to accomplish their respective tasks[1]. Specifically, we use srcML to provide an XML-based markup of C++, thereby simplifying the fact extraction process. First, the tool *cppinc* analyzes source code and generates GraphViz[2]-formatted graphs that can be rendered and inspected to help study the structure of include graphs. Second, another tool *cppconf* analyzes the same source code for macro definition and usage and saves this information for later exploration.

The actual study of the systems proceeded in two phases. In order to study how each of these systems manage portability, we first identified which components of the software are primarily responsible for that concern (i.e., the set of files). Secondly, we extracted and analyzed data about the multi-valued configurable macros. This data is then used to identify emergent patterns for portability and configuration management.

As for identifying the location of portability concerns in software, practical experience allows us to hypothesize about a locus of configuration functionality that we call the *configuration kernel*. To validate this hypothesis, we generated and studied selected include graphs of each library. These graphs reveal a relatively small tree of header files that appear to do little other than conditionally define different macros. Of course, the names of these files are also a great indicator for this kernel since most either contain the word "config", or are nested within a "config" directory.

We then use this set of files in the kernel as the source of configuration data. We processed each of the header files in the kernel, recorded each macro identifier, and where it is used in preprocessor directives (i.e., definitions, undefintions, and conditions). We name these *configurable macros* since the macro identifier can be configured to provide any number of different values.

Our primary classification is based on whether the macro is defined externally (provided by the compiler or build) or internally (defined or undefined within the scope of the studied system). Additionally, lexical analysis (i.e., regular expressions) is used to help identify groups of similarly named macros. We then manually investigated the relationships between the different groups of macros in order to determine if there are any repeated techniques, abstractions, or patterns in the different systems.

3. Examined Systems

We applied this process to three software libraries: the Qt GUI Toolkit[3], the Adaptive Communications Environment (ACE)[4], and the Boost C++ libraries[5]. These libraries were chosen because they are well known, well respected, heavily used, and each is ported to wide variety of systems. For each library, we provide a description of its configuration kernel followed by a classification of the macros appearing within the kernel and a discussion of the configuration architecture.

3.1. Qt GUI Toolkit

The first library we studied is the Qt GUI Toolkit (v. 4.2.2). Qt is a data type library, abstraction provider, and GUI toolkit for Windows, Mac OS X, X11, and embedded (Qtopia) environments. While our study focuses only on the X11 source code distribution, we should note that most of the files in the Windows, Mac, and X11 distributions are identical although windowing-specific implementations of some code (outside the configuration kernel) is provided by the different distributions. The library is segmented into a number of modules, the core of which provides a common set of data structures (e.g., `QList` and `QRect`) and algorithms.

[1] http://www.sdml.info/projects/srcml/
[2] http://www.graphviz.org/

[3] http://www.trolltech.com/
[4] http://www.cs.wustl.edu/~schmidt/ACE.html
[5] http://www.boost.org/

Qt employs a configure script that is used to set install directories, determine buildable components, and configure external dependencies. Most importantly, it is responsible for generating the qconfig.h header file.

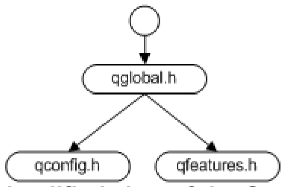

Figure 1. A simplified view of the Qt configuration kernel. The circle indicates an arbitrary inclusion.

Identifying the structure of Qt's configuration kernel is not entirely straightforward. Despite the fact that there are several qconfig-* header files, none of these are ever included. Visual inspection of the include graph reveals a small tree of header files rooted at qglobal.h. Further investigation reveals that this is the root of the configuration kernel. An abridged version is depicted in Figure 1.

Unlike other systems that we studied, Qt embeds most of its configuration in a single header file, qglobal.h. In fact, the contents of qconfig.h and qfeatures.h only define and test macros related to the configuration of external dependencies and the selection of features to build respectively. We used this kernel as the input to *cppconf* to extract information about configurable macros.

Table 1. Nested "namespaces" defining macros in the Qt GUI toolkit. Namespaces are given by regular expressions.

Namespace	Description
^Q_OS	Operating system identities on which Qt can be compiled.
^Q_CC	Compiler identities and versions supported by Qt.
^Q_WS	Windowing systems identities supported by Qt (e.g., Win32, X11)
^QT_MODULE	Defines specific functional subsets of the Qt library.
^QT_EDITION	Defines subsets of modules available for product variants.
^Q.*_EXPORT$	Defines export expressions for C++ class definitions.

Qt references 454 macros in its configuration kernel. Of these, 155 are *configuration primitives* – macros that are defined outside the body of analyzed source code but tested in preprocessor conditions within. Interestingly, we can find two exclusive subsets of this group by recognizing naming conventions for macro identifiers. Macros beginning with underscores are allocated to compiler-specific keywords, identifiers, and extensions, whereas macros starting with Q_ or QT_ are (intuitively) specific to Qt. For example, macros such as __i386__, _WIN32, and __GNUC__ identify hardware architecture, operating system, and compiler, respectively. Likewise, configuration primitives such as QT_BUILD_XML_LIB and QT_CUPS are defined by the build and configuration. The former is defined when the XML implementation is being compiled, and the latter is asserted when Qt is built with CUPS (Common UNIX Printing System) support.

The remaining 299 macros defined (or undefined) within the configuration kernel are of much greater interest because they contribute to the organization of the software. Using the same lexical analysis, we find that most (284) are within the Qt "namespace". Extending this technique to examin e subsequent underscore-separated identifiers, we can easily extract a number of well-defined subsets of Qt-provided macros. These are listed in Table 1.

Another significant set of macros used in preprocessor directives is that which describes specific build features and external dependencies. These macros are easily identified by the use of the _NO_ or _BROKEN_ identifiers in their names. Macros like QT_NO_SLIDER can be used to exclude classes from the library's build, presumably to reduce the memory footprint on embedded devices. Similarly, macros like QT_NO_CUPS are used to determine the presence of external dependencies.

Examining these macros that are used in preprocessor statements, we find interesting relationships to other macros or build options. For example, closer inspection shows that these macros are derived from the assertion of configuration primitives (e.g., _WIN32 or __GNUC__), especially the Q_OS and Q_CC macros. The Q_WS macros are derived mostly from Q_OS macros, but occasionally require tests of other configuration primitives. From this we can infer that the operating system and compiler are configured independently, but the choice of windowing system is dependant up operating system.

Similarly, feature selection via specific feature descriptors is similarly derived. A set of base descriptors is used to activate or deactivate various classes within the library, and the preprocessor is used to manage dependencies between them. For example, the assertion of QT_NO_SLIDER results in the assertion of QT_NO_DIAL and QT_NO_SCROLLBAR as well, effectively removing all related classes from the library. Interestingly, the module and edition macro groups also play a role in feature selection. Specifically, these are used to restrict the build of the library components based on the Qt distribution the developer is using.

Many of the remaining macros seem to be defined for general-purpose programming support. This includes macros for grammar extension (e.g., Q_FOREACH), adapting compiler-specific language extensions (e.g., `Q_DECL_EXPORT`), and preprocessor functions (e.g., `Q_ASSERT`).

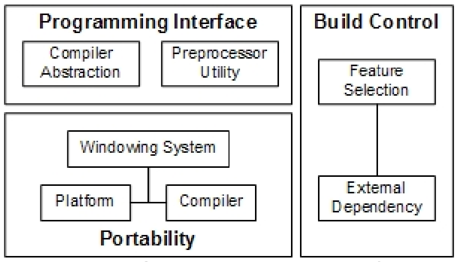

Figure 2. The Qt preprocessor configuration architecture implements multiple concerns in a layered fashion.

Figure 2 depicts an abstracted view of the Qt preprocessor architecture. The library essentially manages two distinct concerns: portability, and build control. The portability concern is entirely managed by macros in the operating system, compiler, windowing system, and their derived macros. The API is built directly on top of portability macros to provide compiler abstractions (e.g., `Q_FOREACH` or `Q_TYPENAME`) and preprocessor utilities for library and application developers (e.g., `Q_GLOBAL_STATIC`).

The build control facility is used to select classes that are compiled into the library at product build-time through both feature descriptors and module/edition specification. Note that it is entirely orthogonal to portability and the API. This is to say that macros in this concern do not generally affect or require the macros in any other.

3.2. Adaptive Communications Environment

The Adaptive Communications Environment (v. 5.5) is a fairly expansive library that provides portable implementations of concurrent and network design patterns. At its core ACE provides a low-level POSIX-like portability layer that reportedly supports compilation on about twenty different operating systems from embedded platforms to Cray supercomputers (excluding individual versions).

ACE has two mutually exclusive methods of configuration. Previous versions of ACE were configured by copying a platform- and compiler-specific header to the `config.h` file. ACE has 89 distinct configuration headers that can be included directly (although this is misleading since some are not intended to be included directly). Newer versions can be configured via an *autoconf*-generated configure script. Rather than using the configure script, we created a `config.h` file that included all 89 distinct configurations allowing us to create a full include graph.

Much like Qt, the ACE configuration kernel is not entirely obvious. Inspecting the include graph reveals that the inclusion of the `config.h` header file occurs only in the head of the `config-macros.h` header file. This is included only by the `config-lite.h` header file, which, in turn, is included by a relatively small set of header files (e.g., `config-all.h`). A simplified include graph is shown in Figure 3. Application-included headers include either the `all` or `lite` configuration variants depending on their requirements. At the backend, our `config.h` header includes 89 different `config-*` header files (depicted by the cloud in Figure 3).

What the graph does not show is the dependencies between compiler and platform header files. Careful inspection shows that platform-specific headers (e.g., `config-linux.h`) will often include compiler-specific headers (e.g., `config-g++-common.h`). Unfortunately, this pattern does not hold for every platform/compiler. More common development platforms such as Linux and Win32 employ this technique to some degree. The SunOS configuration differs significantly in design, in that it the configuration for newer versions include and update older versions' configurations, but does not include any header files outside its own tree.

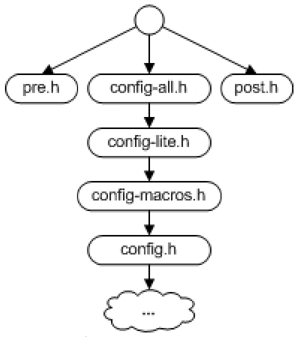

Figure 3. A simplified representation of the ACE include graph. The cloud indicates a number of possible inclusions.

The primary role of this include structure is to define a common platform POSIX-like API upon which all other abstractions are built. In fact, the `config-all.h` header actually includes a significant portion of that API (which is outside the kernel). Most of the configuration macros are defined through inclusions of the `config-macros.h` header. The include graph also has two very obvious files that are included by almost every other include file: `pre.h` and `post.h` header files. These files set compiler options via `pragma` statements for a few different compilers, but do not define or reference any macros. We used the include tree rooted at `config-lite.h` as the input to *cppconf*.

Our analysis of the ACE kernel includes 1412 referenced macros. Of these, 147 were defined externally. Much like Qt, most of the externally defined macros are compiler-provided macros that identify compiler vendors and versions, operating systems, and computer hardware. There are few (27) that are specific to ACE. Some of these macros, such as `ACE_BUILD_DLL` and `ACE_HAS_VALGRIND` are defined during the build or to assert the availability of external dependencies.

The remaining 1265 macros are all defined within the scope of ACE configuration kernel. While not as clearly segregated as the identifiers in Qt, we can still identify a number of logical groupings by use if not by name.

- *System identities* – There are many ACE macros that identify different compilers and operating system. The `ACE_CC` namespace contains all compiler information, but each operating system is given its own set of identifiers (e.g., `ACE_PSOS`, `ACE_WIN32`).
- *Inclusion Controls* – Some (but not all) of the configuration files use header guards to prevent redundant inclusions.
- *Constants* – The configuration kernel defines a number of constants. One source of these is POSIX error codes (`E*` constants) defined to supplement incomplete standard libraries. Another is the size of built in types (i.e., integers).
- *Function mapping* – ACE defines a number of macros that expand to the names of functions of specific runtime libraries. Examples include `ACE_TEXT` and `pthread` macros that expand to a collection of Win32 calls and different *pthread* names respectively.
- *Compiler Abstraction* – A relatively small set of macros is provided as part of the API. Macros like `ACE_EXPLICIT` and `ACE_MAIN` abstract C++ keyword variations and program entry points.

Just like Qt, the majority of internally defined macros are feature descriptors, all prefixed with `ACE_HAS`, `ACE_LACKS`, or `ACE_NEEDS`. In stark contrast to Qt, where feature descriptors are asserted to exclude classes from a build, these identifiers play an integral role in the construction of the POSIX adaptation layer. These feature descriptors are used to describe the presence of specific header files (e.g., `ACE_HAS_SELECT_H`), functions (e.g., `ACE_LACKS_DUP2`), types (e.g., `ACE_HAS_SSIZE_T`), and various external dependencies (e.g., `ACE_HAS_SSL`).

The relationships between these macros are interesting. Expectedly, we find that the sysstem identities are derived immediately from configuration primitives. Perhaps even more interesting is that the definition of feature descriptors are (for the most part) also immediately derived from configuration primitives. For example, in the common Linux header, `ACE_HAS_SNPRINTF` is defined if one of a number of externally defined macros is asserted (e.g., `_BSD_SOURCE` or `_XOPEN_SOURCE`). From this, we can assume that there is a high level of coupling between the operating system, compiler and standard runtime libraries.

Figure 4 depicts a generalized view of the ACE configuration architecture. Here, the portability concern is managed by a trinity of three tightly coupled concerns aspects: the platform, the compiler, and the feature specification of the C runtime. Interestingly, the only discernable pattern in the configuration kernel appears to be the absence of internal abstraction or layering within the portability concern. The identifiers defining compiler, platform and features are referenced only sparingly within the kernel itself. In fact, only 166 internally defined, non-header guard macros are actually tested within the kernel. However, the features identified in the portability concern are used to define a POSIX adaptation layer via constant definition and function mapping. Also include in this API is a set of macros for abstracting compiler deficiencies and differences.

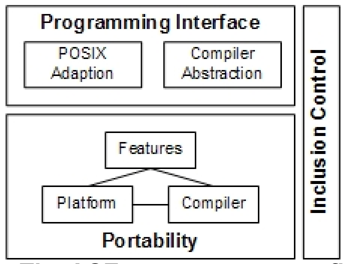

Figure 4. The ACE preprocessor configuration architecture.

The only orthogonal concern represented by macros in the configuration kernel is inclusion control, specifically the protection of redundant header inclusion during preprocessing.

3.3. Boost C++ Libraries

The Boost C++ Libraries (v. 1.33.1) is a large set of C++ libraries that supply data structures and algorithms for application developers. One purpose for the Boost Libraries is to act as a testing ground for components that may eventually become a part of the Standard Template Library (STL). Unlike ACE and Qt, the Boost libraries do not supply a low-level interface, but instead relies on the correctness of vendor-supplied implementations of the STL. In cases where platform-specific API's are required, they are deeply hidden by abstraction. The Boost Filesystem library is typical of this design.

Like ACE and Qt, Boost also has a configure script. However, the configure script is only used to set build variables related to external dependencies (e.g., Python), install directories, and build profiles (i.e., debug/release and threading). Fortunately, we do not need to run the configure script because most of the configuration is internal to the Boost libraries.

The Boost configuration kernel is relatively easy to identify. From previous experience, we know that Boost is configured through its Boost Config library – a set of header files that selects the correct compiler, standard library and operating system at compile time. The entire configuration is included through the `config.hpp` header file, which in turn includes a number of files in the `config` directory. We validated that this is the configuration kernel by examining the include graphs of several Boost libraries.

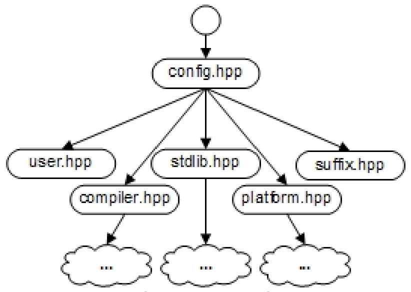

Figure 5. A simplified view of the Boost include graph. Actual file names are truncated for brevity. Clouds indicate a number of possible inclusions.

Figure 5 depicts a simplified version of the Boost configuration kernel. Boost also provides facilities for changing ABI (Application Binary Interface) flags in a similar manner to compiler, platform and standard library configuration. There are also other include files in the configuration kernel that auto linking features and determining POSIX features. These have been omitted from the diagram for simplicity.

As mentioned, Boost is unique (among studied libraries) in its configuration for a number of reasons – much of which is related to the use of macro expansion to include files. For example, the so-called user include file can be defined at compile time to reference a developer-specific file by defining a macro, `BOOST_USER_CONFIG`, to reference a specific file (e.g., "`my_config.hpp`") although it defaults to a mostly empty `user.hpp`. The user defined file can set the values of other include macros to override the default configuration by defining alternative *include macros* – those which are expanded to identify alternative include files. Examples, include `BOOST_COMPILER_CONFIG` and `BOOST_STDLIB_CONFIG`. If undefined, they reference default include files.

Not surprisingly, Boost's default include files actually act as selection functions for system dependencies. For example, the default include file for compiler selection is the `select_compiler_config.hpp` header file. These selectors exist for both standard libraries and platforms as well. When included, selectors will test configuration primitives (much like a C `switch` statement) to determine the vendor of the specific dependency. Identification results in the definition of an include macro that references a pre-implemented configuration for the resource.

Also unique is the fact that the kernel physically separates concerns in its directory structure. Within the `config` directory, we can find subdirectories containing header files for 16 different compilers, 11 different platforms, and 9 different standard template library implementations. Default include macros reference files in these directories (e.g., `gcc.hpp` and `stlport.cpp`). Contrast this with Qt, which primarily uses a single, large configuration header, and ACE, which has a flat, unstructured collection of configuration headers.

The suffix header performs post-selection configuration. This is to say that it defines many of the macros that identify capabilities and deficiencies appearing in different compilers, platforms and standard libraries.

There are only 339 macros referenced in the Boost configuration kernel. Of these, just over half (185) are defined externally. Unlike both Qt and ACE, where a significant number of these macros within the library's namespace, there are only 9 in Boost and all but one are obviously related to controlling the preprocessor configuration (e.g., `BOOST_NO_CONFIG` causes Boost to skip most of its configuration headers). The remaining externally defined macros identify compilers and operating systems, and aspects of standard template library implementations.

The remaining 154 internal macros fall into the following groups:

- *Configuration control* – Macros identifying include files (e.g., `BOOST_USER_CONFIG`) and inclusion logic (e.g., `BOOST_ASSERT_CONFIG`).
- *External identities* – Macros identifying the compiler, standard library, and operating system (e.g., `BOOST_MSVC` and `BOOST_PLATFORM`).
- *Programming support* – Boost provide a number of different macros that abstract compiler workarounds or differences in syntax and semantics (e.g., `BOOST_STATIC_CONSTANT`). This also includes utility macros such as `BOOST_JOIN` and `BOOST_STRINGIZE` which implement the `#` and `##` operators.

However, most of the internally defined macros are feature descriptors of the form `BOOST_HAS` or `BOOST_NO`. Unlike Qt, which used these to except classes from the build and ACE, which used these to describe POSIX-like features, Boost primarily uses these to describe compiler and STL deficiencies. For example, the `BOOST_NO_SFINAE` is asserted for compilers that do not correctly implement argument substitution for templates, and `BOOST_HAS_HASH` is asserted when a standard library provides implementations of the `hash_set` and `hash_map` classes.

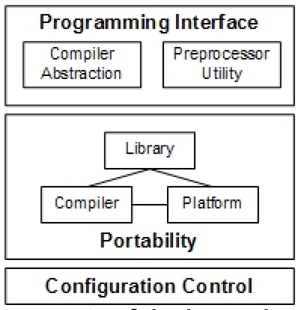

Figure 6. Components of the layered architecture for configuring the Boost C++ Libraries.

Figure 6 depicts a generalized architecture for the configuration kernel of the Boost C++ Libraries. The bottom-most layer, configuration control, defines macros that are used to control the configuration process. The primary portability concern is similar to that of ACE in that it does not build on internal abstractions. but couples the evaluation of configuration primitives to build a single configuration for the compiler, standard library and platform. In Boost, the most user-visible macros are used for mitigating compiler variations and some simple macros for stringification and token pasting. These macros comprise the programmer's interface – the API.

4. Emergent Patterns in Preprocessor Usage

From our study of the three systems, we observed a number of specific techniques employed to help increase portability and reduce maintenance effort. Not all systems use (or even subscribe) to all of these patterns, yet they appear frequently throughout systems and as such warrant clear description as the represent common solutions to portability and configuration management.

4.1. Logical Namespaces

The first and most obvious pattern that emerges is the use of naming conventions for macro identifiers. While this is not what one might typically call a design pattern or even programming idiom, naming conventions provide a simple mechanism for separating concerns in the absence of lexical scoping. Macro identifiers are typically grouped into what we term *logical namespaces* by using common identifier "stems", separated by underscores. For example, the `Q_CC_GNU` macro is in logical namespace compiler (`CC`) nested within the logical namespace Qt (`Q`). Consistent and thoughtful scoping of macro identifiers can greatly ease program comprehension efforts, and therefore reduce maintenance cost by reducing time spent understanding the system.

A thorough usage of this technique is shown in Table 1 – the logical namespaces used in Qt's configuration. Unfortunately, only Qt implements namespaces to any great extent, and only then for a limited number of concepts. Boost and ACE are somewhat less consistent with their use of namespaces, but do use the technique to help segregate feature descriptors (i.e., the `HAS`, `NO`, and `LACKS` macros) from other concerns.

4.2. Replaceable and Parameterized Inclusion

When porting a library or application, maintenance requires adapting the library to a new compiler or platform by modifying the configuration kernel to introduce the new requirements. With open source licensing this can be problematic since modifications to open source products carry (at the very least) documentary and redistribution requirements. When commercial licenses and especially non-disclosure agreements enter the mix, modifying the original source code is out of the question.

Both ACE and Boost have implemented defensive mechanisms for configuration extension. ACE allows developers to create custom configuration headers that are, by definition, not part of the original source code. This replacement scheme requires developers to supply a file or symbolic link with a specific name (i.e., `config.h`). This is then incorporated into the configuration through the kernel's standard include paths.

In contrast, Boost uses a parameterization scheme to accomplish the same task. Here, users can define macros through the build environment that reference user-defined header files. These macros effectively act as parameters to configuration. Boost also extends this technique to include other parameters that allow a configuration to be validated (i.e., an assertion mode) that causes errors when unknown configurations are encountered.

By allowing developers to override a library's default configurations, new platforms can be integrated and tested piecewise without having to interweave new configurations with existing ones. This allows new preprocessor code to be integrated into the system more cautiously, potentially reducing configuration conflicts and bugs.

4.3. Compiler Abstractions

Despite the standardization efforts of the C and C++ programming languages, compiler implementations can vary greatly due to differences in language versions, broken or unimplemented language features, and grammar extensions. Also, major compiler releases occur every couple of years, requiring applications to evolve with them. In order to mitigate the unrelenting progress of compiler innovation, developers must build their software to support both legacy and newer version of a compiler in parallel.

To this end, heavily ported libraries with long life spans (such as those studied here) must accommodate not only multiple versions of the same compiler, but also multiple compilers. This is invariably done using the preprocessor. There are three distinct categories of compiler abstractions: *workarounds*, *grammar adapters*, and *syntax extension*. Workarounds are generally function-like macros that expand to different pieces of code under different compilers to help implement unsupported or broken features. Grammar adapters are similar to workarounds but tend to address version-specific additions or compiler-specific extensions to a language. Syntax extensions are macros that appear to add new keywords or syntax to a language. The most common example of this is the provision of a `foreach` construct.

Using the preprocessor for compiler abstractions is somewhat of a double-edged sword for maintainers. While these can allow the software to be compiled under numerous regimes, the resulting source code can become significantly less readable. Moreover, long-term support for legacy compilers may ultimately reduce the portability of the application, preventing it from introducing new language features or libraries. Also, deciding to remove support for older compilers requires developers to decide whether or not to remove the abstractions, and if so, spend the time doing it.

5. A Common Configuration Architecture

All of these libraries exhibit some form of a layered architecture in the structure of their configuration. Specifically, configuration primitives are used to build internal abstractions that are used by library and application developers. However, due to the unstructured scoping of preprocessor macros, this architecture is "relaxed", meaning that layers and separation are not clearly defined, and that any client can access information or functionality at every level. Figure 7 depicts the common configuration architecture exhibited by the configuration kernels of the studied systems.

This architecture consists of several different layered and orthogonal concerns: configuration, portability, inclusion control, build control, and the programming interface. The lowest layer is that of configuration control – the use of macros to control the include process and thereby controlling which configuration headers are incorporated into the binary. The orthogonal concern of Inclusion control defines the set of macros and conditions used to ensure that files are included in an appropriate manner (typically once). Likewise, build control is the set of directives responsible for selection of code corresponding to some build configuration option (i.e., define `QT_NO_SLIDER` to deselect the compilation of the QSlider class).

The portability concern is comprised of two distinct sub-layers: a portability core, and derived observations. The core relies on configuration primitives provided by two (mostly) independent dependencies: the compiler and the platform. Here, the platform is taken to be the architecture, operating system, and C runtime library. In the second layer, configuration wrappers are used to identify, the vendor and version of the compiler, platform, architecture, etc (i.e., external identities). Feature descriptors provide a detailed specification of those dependencies, usually in terms of capabilities and deficiencies (i.e., `BOOST_HAS_THREADS` or `ACE_LACKS_UNLINK`).

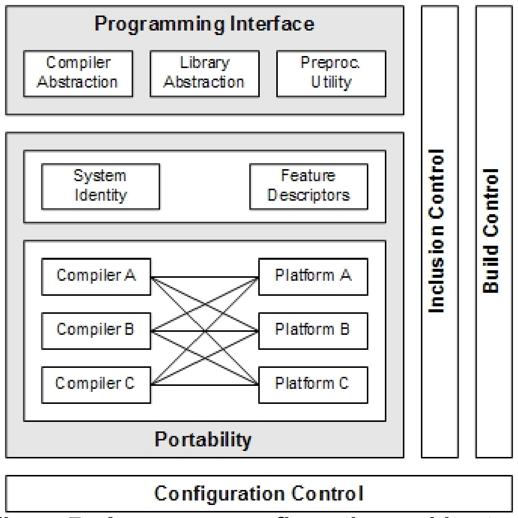

Figure 7. A common configuration architecture depicting the primary layered and orthogonal concerns.

The API, or the top-level layer in the architecture, builds on the macros defined in the portability concern to abstract or mitigate differences between the underlying dependencies. Compiler abstractions build workarounds for broken compilers, grammar abstraction for different versions of the language, and syntax extensions for programmer convenience. Library adaptation provides similar features for the core or abstracted library. This feature is primarily used to address differences in compliance between implementations of a common library (e.g., POSIX or the STL). Finally, preprocessor utilities are just that: macros intended for programmer utility such as stringification, token pasting, and code generation for repetitive programming tasks.

6. Build Configuration Software

Each of these libraries is distributed with an automatic configurator that can generate a pre-build, platform-specific configuration. Qt's configuration script generates the actual configuration header to define macros that identify the presence of external dependencies (e.g., various X11 extensions). Boost's configuration script does exactly the same thing with Python and ICU (Unicode) libraries. Here, we find two examples of external programs used to control the specification and versioning of (optionally) required libraries. However, neither of these programs attempts to determine the configuration of the host platform and compiler. The preprocessor is responsible for these configurations.

ACE is also distributed with an *autoconf*-generated configuration script. Unlike the previous scripts, this will attempt to deduce sets of feature descriptors for every external dependency – including those typically covered by preprocessor-based configurations.

While it appears that configurators are useful for managing build options related to 3rd party dependencies such as Python in Boost and numerous X11extensions for Qt, there is no conclusive evidence, despite widespread acceptance of the GNU *autotools*, that total automatic configuration is a successful strategy for increasing portability or reducing maintenance effort. In fact, it might be observed that introducing a new compiler, platform or external dependency is actually *more* complicated since it requires learning the language and tooling of the configurator. Moreover, the use of configurators still results in massive numbers of configuration macros that must still be tested in the application in order to be useful. These macros now also accrue the added cost of being untraceable without examining the configurator.

7. Related Work

Despite pervasive use of the C preprocessor, literature (both research and practical) is quite rare when compared to the volumes written on the languages whose existence depends upon it. The first case study of preprocessor usage with respect to portability was given by Spencer et al. [17]. This study analyzed preprocessor usage in the C News program as it evolved. The study suggests pragmatic approaches to using the preprocessor, but does not provide, what we think of today, as pattern for its applications.

The only previous comprehensive study of preprocessor usage was conducted by Ernst et al. in [4] using PCp^3 [1] to analyze the incidence of preprocessor usage in a number of C programs. This empirical data is used to create taxonomies of macro definitions and usage. Specifically, this work discusses classifications of macros based on the structure of their definition, their extra-linguistic capabilities, (potentially) erroneous definitions, and multiple, different, and inconsistent definitions. Macro usage is discussed in terms of frequency, context, and usage in preprocessor conditions.

Most relevant to our work, the study categorizes macros used in conditions as being used for one of several concerns. Our investigation mostly supports this taxonomy. There are, however, some differences due to the fact we studied the intent of usage rather than deriving a classification from name, definition, and context. The biggest difference comes from the classification of macros concerning the portability of machine, library, and language, and additionally macros labeled as "miscellaneous". In [4] the classification "machine portability" encompasses macros defining the architecture and hardware, language whereas "library portability" encompasses externally defined macros, and "miscellaneous" represents reserved identifiers (macros of the form __XXX__). Our analysis clearly shows that these macros actually describe the operating system, compiler, and versions thereof. Moreover, externally defined macros can be used to control the build and are not directly related to portability, but are more likely to be related to general configuration.

In another (albeit very different) approach to preprocessor analysis, formal concept analysis is applied to conditionally compiled regions of source code to investigate the relational structure of configurations in [10]. Although a completely different approach, this approach is useful for showing the relation between configuration macros, via the code configured by them. This was later used to suggest configuration restructuring via operations on concept lattices [16].

Otherwise, there have been a number of approaches to preprocessor analysis for a number of applications. For example the preprocessing language has been treated as a program dependency graph (PDG) to support abstract program analysis and configuration reachability [5, 6, 9, 11, 12]. Similarly, in [19] preprocessor directives are modeled for both static and dynamic analyses. There have also been a number of approaches to preprocessor analysis for reengineering, restructuring, and refactoring [2, 7, 8, 14, 18, 20].

8. Conclusions and Future Work

We examined three heavily ported C++ libraries and their use of the C preprocessing language to manage portability concerns. Despite the absence of reliable design documentation or even descriptive literature (e.g., books on programming), we find emergent patterns in C preprocessor usage. Specifically, we identify four preprocessor techniques that contribute to effective portability management. Specifically we find that the use of logical namespaces, replaceable and parameterized inclusions, macro-level compiler abstractions all contribute toward reliable software porting. Most importantly, we show that in long-lived and portable systems these concerns are all managed within a common configuration architecture. This common architecture acts as a model for developers creating portable software.

We believe that this work only scratches the surface of preprocessor and source code configuration analysis. We believe that our method for modeling include-files and especially configuration macros is capable of supporting much more detailed analysis. We intend to use this approach to model the relationships between macros and the relation between preprocessor directives and the source code itself. We believe that these techniques will contribute to the creation of refactoring tools for portability and configuration management as well as a better understanding of software evolution from the perspective of adaptability and portability.

To conclude, we do not foresee a future lessening of preprocessor usage. In fact, we predict that portability via the preprocessor will become increasingly important with new language features for both C and C++ on the horizon. Moreover, the platforms on which these programs compile will also continue to evolve. Any long-lived and heavily ported library will have to contend with the multitude of both new and old versions of compilers, platforms and other libraries.

9. References

[1] Badros, G. J. and Notkin, D., "A Framework for Preprocessor-Aware C Source Code Analyses", Software: Practice and Experience, vol. 30, no. 8, Jul 2000, pp. 907-924.

[2] Baxter, I. D. and Mehlich, M., "Preprocessor Conditional Removal by Simple Partial Evaluation", in Proceedings of 8th Working Conference on Reverse Engineering (WCRE '01) Stuttgart, Germany Oct 2-5 2001, pp. 281-290.

[3] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

[4] Ernst, M. D., Badros, G. J., and Notkin, D., "An Empirical Analysis of C Preprocessor Use", IEEE Trans. on Software Engineeering, vol. 28, no. 12, Dec 2002, pp. 1146-1170.

[5] Favre, J.-M., "Preprocessors from an Abstract Point of View", in Proceedings of International Conference on Software Maintenance (ICSM '96), CA, Nov 4-8 1996, pp. 329-339.

[6] Favre, J.-M., "Understanding in the Large", in Proceedings of 5th International Workshop on Program Comprehension (IWPC '97), Dearborn, Michigan, Mar 28-30 1997, pp. 29-38.

[7] Garrido, A. and Johnson, R., "Refactoring C with Conditional Compilation", in Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE '03), Montreal, Canada, Oct 6-10 2003, pp. 323-326.

[8] Garrido, A. and Johnson, R., "Analyzing Multiple Configurations of a C Program", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM '05), Budapest, Hungary, Sep 25-30 2005, pp. 379-388.

[9] Hu, Y., Merlo, E., Dagenais, M., and Lagüe, B., "C/C++ Conditional Compilation Analysis using Symbolic Execution", in Proceedings of International Conference of Software Maintenance (ICSM '00), San Jose, CA, Oct 11-14 2000, pp. 196-206.

[10] Krone, M. and Snelting, G., "On the Inference of Configuration Structures from Source Code", in Proceedings of 16th International Conference on Software Engineering (ICSE'94), Sorento, Italy, May 16-21 1994, pp. 49-57.

[11] Latendresse, M., "Fast Symbolic Evaluation of C/C++ Preprocessing Using Conditional Values", in Proceedings of 7th European Conference on Software Maintenence and Reengineering (CSMR '03), Mar 26-28 2003, pp. 170-179.

[12] Latendresse, M., "Rewrite Systems for Symbolic Evaluation of C-like Preprocessing", in Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR '04), Tampere, Finland, Mar 24-26 2004, pp. 165-173.

[13] Maletic, J. I., Collard, M. L., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 289-292.

[14] Mennie, C. A. and Clarke, C. L. A., "Giving Meaning to Macros", in Proceedings of 12th IEEE International Workshop on Program Comprehension, Italy, Jun 24-26 2004, pp. 79-85.

[15] Parnas, D. L., "Software Aging", in Proceedings of International Conference on Software Engineering (ICSE), Sorrento, Italy, May 16-21 1994, pp. 279-287.

[16] Snelting, G., "Reengineering of Configurations based on Mathematical Concept Analysis", ACM TOSEM, vol. 5, no. 2, Apr 1996, pp. 146-189.

[17] Spencer, H. and Collyer, G., "#ifdef Considered Harmful, or Portability Experience with C News", in Proceedings of 1992 Summer USENIX Conference, San Antonio Texas, Jun 1992, pp. 185-198.

[18] Spinellis, D., "Global Analysis and Transformations in Preprocessed Languages", IEEE Transactions on Software Engineeering, vol. 29, no. 11, Nov 2003, pp. 1019-1030.

[19] Vidács, L., Beszédes, Á., and Ferenc, R., "Columbus Schema for C/C++ Preprocessing", in Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR '04), Mar 24-26 2004, pp. 75-84.

[20] Vittek, M., "Refactoring Browser with Preprocessor", in Proceedings of 7th European Conference on Software Maintenence and Reengineering (CSMR '03), Mar 26-28 2003, pp. 101-110.