The Build-Time Software Architecture View

Qiang Tu and Michael W. Godfrey
Software Architecture Group (SWAG)
Department of Computer Science, University of Waterloo
email: {qtu,migod}@swag.uwaterloo.ca

Abstract

Research and practice in the application of software architecture has reaffirmed the need to consider software systems from several distinct points of view. Previous work by Kruchten [9] and Hofmeister et al. *[6] suggests that four or five points of view may be sufficient: the logical view (*i.e., *the domain object model), the (static) code view, the process/concurrency view, the deployment/execution view, plus scenarios and use-cases. We have found that some classes of software systems exhibit interesting and complex* build-time *properties that are not explicitly addressed by previous models. In this paper, we present the idea of build-time architectural views. We explain what they are, how to represent them, and how they fit into traditional models of software architecture. We present three case studies of software systems with interesting build-time architectural views, and show how modelling their build-time architectures can improve developer understanding of what the system is and how it is created. Finally, we introduce a new architectural style, the "code robot" that is often present in systems with interesting build-time views.*

1 Introduction

The recently emerged field of software architecture has attracted tremendous interest from researchers in both the academic and industrial communities. Perry and Wolf predicted that the 1990s would be the decade of software architecture [11]; it seems clear that this prediction was accurate and that research into software architecture will continue to mature and evolve for several years to come.

As an area of study, software architecture is still in its early stages. Several definitions of software architecture have been proposed [11, 13, 2], various architectural styles have been documented [13], and different taxonomies of architectural views have been suggested [9, 6]. However, we have found that some classes of systems exhibit interesting properties that are apparent only at system build-time. Consequently, we have investigated modelling these properties and incorporating these models into existing taxonomies of architectural views.

The structure of the rest of this paper is as follows: Section 2 summarizes previous definitions of software architecture and taxonomies of architectural views; it also introduces the idea of build-time architectural views. Section 3 presents three case studies of software systems that have interesting build-time architectural views: the GCC compiler family, the Perl language interpreter/run-time system, and the Java Native Interface (JNI). Section 4 discusses the relationship between build-time views and configuration and build management and presents an informal metamodel for build-time architectural views. Section 5 introduces the "code robot" architectural style, which is common to systems that have interesting build-time views. Finally, Section 6 summarizes the work presented here.

2 Software Architecture

Various definitions of software architecture have been suggested by researchers. Bass *et al.* define software architecture as "the design and implementation of the high-level structure or structures of the system. It comprises software components, the externally visible properties of these components, and relationships among them". Perry and Wolf presented their definition of software architecture as a mathematical model:

$$\textit{Software architecture} = \{\textit{elements, form, rationale}\}$$

Others have continued to refine and adapt the definition. For example, Boehm uses the term "connections" instead of form, and supplements constraints to the rationale. Shaw and Garlan define software architecture as components, connectors, and configurations [13].

2.1 Architectural Views

The "system" modelled by software architecture is usually large and complex in nature. Different stakeholders (*e.g.,* users, business analysts, programmers, testers, and maintainers) typically care about only specific aspects of the software system, and have widely varying mental models based on how they experience the system. To separate the different areas of concerns, and to reflect the dynamic nature of software architecture, the notion of "views" of software architecture is introduced.

Kruchten proposed a "4+1" view model to describe the software architecture of a system from multiple perspectives, to separate concerns from various stakeholders of the architecture, and to differentiate the functional and non-functional requirements [9]:

- The *logical view* is the analyst's abstract object model, which captures the abstraction, encapsulation, and inheritance in the problem domain.
- The *process view* models concurrency and synchronization aspects of the software system.
- The *physical view* maps the software elements onto the hardware execution environment; this view also capture the distributed aspects of some systems.
- The *development view* models the the static organization of the software in its development environment. It mainly focuses on the source code structure.
- *Scenarios* and use-cases show how all four views work together to satisfy user requirements.

Hofmeister *et al.* have defined a similar taxonomy, which we shall refer to as the "four views" model [6]:

- The *conceptual architecture view* is analogous to the "4+1" logical view.
- The *module architecture view* shows the high-level design of the system. Its purpose is twofold: it describes how to map the conceptual view onto the high-level design artifacts, and it describes the layering structure of subsystems and specifies the interfaces between subsystem layers and low-level design artifacts (*e.g.*, source code modules).
- The *code architecture view* is the low-level design view that describes the interfaces and interdependencies between source code, files, directories, libraries, *etc.*
- The *execution architecture view* describes the structure of a system in terms of its run-time platform elements, such as tasks, processes, threads, and address spaces. This view captures the distributed and concurrent aspects of the system, as well as other non-functional requirements such as resource sharing, scheduling policies and load balancing.

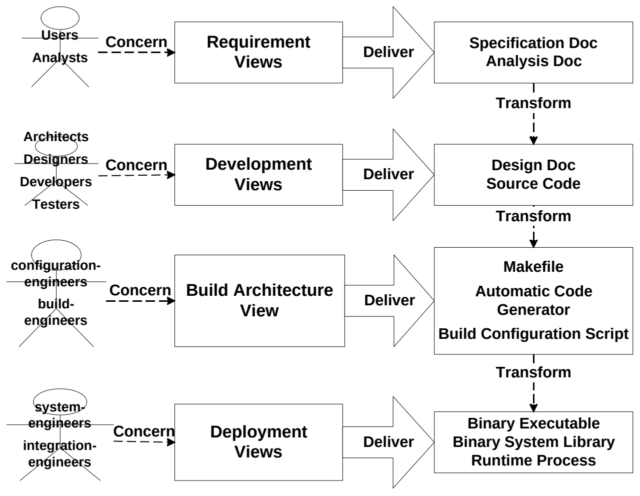

Figure 1. Transformation between architecture views.

Roughly speaking, we can consider that the above models concern three kinds of views: requirements views (*logical*, *process*, *scenarios*, and *conceptual*), development views (*conceptual*, *module*, and *code*), and deployment views (*physical*, *execution*). However, we have noticed that many software systems exhibit interesting structural and behavioural properties that are apparent only at system configuration and build-time, and that these properties are not explicitly considered by either the "4+1" or "four views" models. Therefore, we now examine the idea of *build-time architectural views* in more detail.

2.2 Build-time Architectural Views

Once a software system has been designed and implemented, it must be configured, compiled, and linked for a particular environment before it can be deployed. For a small software system written for a single platform, the `make` utility and a single `Makefile` is often sufficient to manage system building. However, for systems that are large and complex, that run on a variety of platforms, and that support multiple functional configurations, build management is non-trivial. For example, the Perl build process has become so complicated that a separate development effort has been created just to write the configuration scripts and `Makefile` for each release [14]. Furthermore, the GCC system (GNU Compiler Collection) exhibits even more dynamic and interesting build-time behaviours, including as multiple passes of compilation and a significant amount of automatic source code generation.

For such large and complex systems, build management is typically performed by dedicated employees ("build engi-

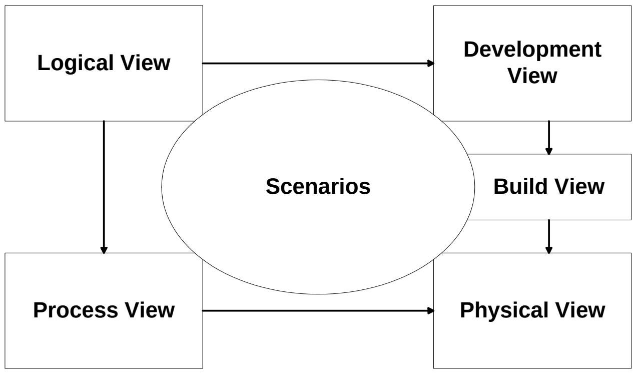

Figure 2. Kruchten's "4+1" model enhanced with the build-time architectural view.

neers") using a specialized build management system. The main responsibilities of a build management system are to configure and manage the build scripts, to provide the ability to define repeatable system build procedures, to maintain consistency in system builds, and to control the various build tools that are involved in the process.

Build-time architectural views should capture configuration and built-time properties that are extractable from build management artifacts, such as build scripts, the source and object files, and configuration choices. At the very least, they should model the compilation dependencies among compilation units, time-sequence configuration of the compilation procedure, and source code generation at build-time.

As mentioned above, neither the "4+1" or "four views" models explicitly addresses the idea of build-time architectural views. Figure 1 shows the relationship between build-time views and the other kinds of architectural views, and Fig. 2 shows how the "4+1" model may be extended to include build-time views.

As shown in Fig. 1, a software system is transformed from one kind of architectural view to another as the result of development and deployment efforts. For example, the software designer studies the logical view of the software system, and applies various design techniques to layout the development view. When the development phase is complete, the build engineers write the `Makefile` and actually build the system from a collection of source code files (development view or code architecture view) into a suite of executables and libraries (execution architecture view or physical view) that work together closely to deliver the functional and non-functional requirements demanded by the customer

Having outlined what we consider build-time architectural views to be and how they fit into traditional models of software architecture, we now examine as case studies three software systems that have interesting build-time views: GCC, Perl, the use of the Java Native Interface (JNI).

3 Case Studies

3.1 Build-Time Architecture of GCC

The GNU Compiler Collection (GCC) is a suite of compilers that supports several programming languages (C, C++, Object-C, *etc.*) and runs on a variety of hardware architectures and operating systems [7]. GCC is an open source software system: the source code is freely available, and it is therefore possible for users to compile and install their own customized versions. In fact, portability and customizability are major design goals for GCC, and this has had a significant influence on its software architecture.

In examining the GNU C compiler (`gcc`), a subcomponent of GCC, we have found that it exhibits two particularly interesting phenomena at build time: it uses a multiple-pass compilation process (also known as "bootstrapping"), and it automatically generates significant portions of the system source code.[1] In this section, we will try to capture these phenomena with a formalized build-time architectural views, which will clearly define the components, form, and rationale. The GCC version used in this case study is 2.7.2.3.

3.1.1 Bootstrapping in `gcc`

Bootstrapping is a common technique employed in compiler design to "use the facilities offered by a language to compile itself" [1]. A typical bootstrapping process consists of several steps [12]:

1. Compile a reduced version of the compiler in a different environment.

2. The first reduced compiler is used to compile itself for the target machine.

3. The compiled compiler then compiles itself on the target computer.

4. The language is enhanced to its full capability and the enhanced version recompiled.

5. If code optimization was left out in previous step, then subsequent recompilation will improve the compiler's own performance.

[1]Note that we use the term "GCC" when referring to the full compiler suite, and we use the term "`gcc`" when referring only to the C compiler component of GCC. Although confusing, this convention is the standard one.

A full-scale bootstrapping is essential for developing the first compiler of a new programming language, or to support a new hardware architecture. More commonly, a simplified bootstrapping procedure is used to develop a new version of the same compiler system. Bootstrapping also provides an excellent opportunity to look for bugs in the compiler, since the compiler source code is usually much larger and more complex than any regression test suite.

In many compiler systems, the only artifact that documents the bootstrapping procedure and configuration is the `Makefile`. However, the `Makefile` itself is usually inscrutable to humans due to the size and complexity of the system, and because it is usually highly templated and parameterized to account for cross-platform issues. The `Makefile` is expanded at build-time after configuration tools have collected information about the hardware and operating system then tailored the `Makefile` for this specific environment. As a result, the only practical way to infer the behaviour that occurs during bootstrapping is to log the execution of `make` at build time, and then to review the log file.

The build-time behaviour of GCC during bootstrapping is shown in Fig. 3. During the bootstrapping process, three different GCC compilers are built. The first one is built by the default system C compiler and linker, and the remaining two are built by GCC itself. In all three builds, the same source files are compiled.[2] Three copies of the GCC compiler executables `gcc` are created at different time and each but the last is immediately used to compile the next version.

First, the existing C compiler on the build platform is used to compile the GCC driver, the C language compiler and the GCC libraries (not the C system library, but the supporting routines for other parts of GCC) from the GCC source code. When the first build is completed, we have an intermediate GCC C compiler that is fully functional. We call it "stage 1" GCC.

For the second build, we run the product from last build, the "stage 1" GCC, on the same GCC source code again. This time, we compile not only the GCC driver, the C compiler, and the GCC libraries, but also the C++ compiler, the Object-C compiler and their supporting libraries. At this moment, all the functional components of GCC 2.7.2.3 (C, C++ and Object-C compilers) are built and integrated with a unified compiler driver `xgcc`. We call this intermediate compiler suite "stage 2" GCC.

"Stage 2" GCC is still not the final product. We run it over the GCC source for the third and the last time. The product of this round of build should be identical to the "stage 2" GCC, because they are complied from the same source code and both by the GCC 2.7.2.3 C compiler. The purpose of this build is for self-testing only: if "stage 2" GCC and "stage 3" GCC are different, this implies that something has gone wrong with the compilation and that there are still some bugs to be worked out.

[2] The `gcc` compiler supports various extensions to the C language that are not part of the ANSI C standard, such as comments that begin with "`//`". Many systems, such as the Linux kernel, take advantage of these language extensions. However, since GCC is usually first compiled by a non-GCC C compiler, the GCC system itself is written entirely in ANSI C.

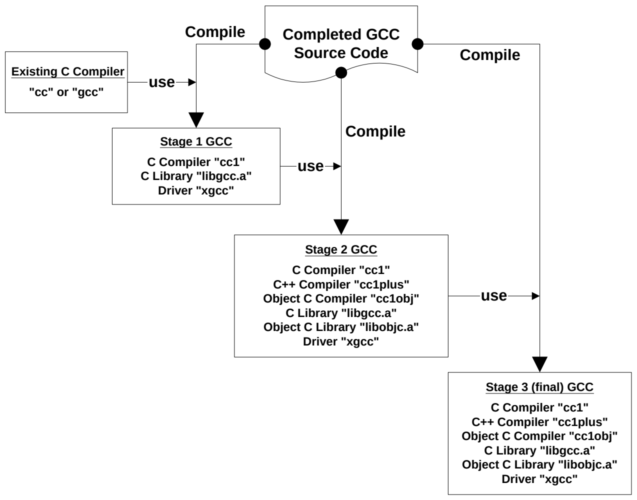

Figure 3. Build-time view of GCC bootstrapping.

The build architecture view of GCC is shown in Fig. 3; the details were manually reverse engineered from the log file of an actual build of GCC 2.7.2.3 on a Sun Solaris 2.6 server with the bootstrapping build option turned on.

The build architecture view captures the dynamic behaviours and configurations of the build process of GCC. The constructs of the view, such as boxes and arrows, have different semantics from other architectural views we have been familiar with. For example, in the traditional module view, the software system is represented as a static structure of a group of programming modules, while the links between components (modules) represent the semantic dependencies and run-time interactions. In the process view, the system is represented as a snapshot of a running organism with different processes collaborating with each other. However, the build architecture view is much more similar to a time-space diagram. The components inside the view do not co-exist at the same moment. The sequence of event happens in the order of from top to the bottom, and from the left to the right. The components, which should be interpreted as either static data or execution units, came into existence in sequence along with the events that modelled as arrows in the view. The arrows between components also represent the build-time (dynamic) dependencies between compilation entities.

3.1.2 Automatic Code Generation During Build-Time

In this section, we will discuss another interesting aspect of the build architecture of GCC, and its role in the architectural transformation of system views.

The build process of GCC consists of two major activities: `configure` and `make`. The "`configure`" process first checks whether the build environment is supported by GCC. Next, `configure` probes the build environment for information such as the CPU architecture, operating system, available system libraries, *etc.* Based on the probed information, it generates the `Makefiles` using the templates and system configuration information that `configure` just collected. In the "`make`" phase, the compilation of the source, linking of object file and installation of final binaries are performed.

For many software systems, including the Linux kernel, Mozilla, and VIM whose architectures we have analyzed before [3, 5, 15], the build process is fairly straightforward: virtually all of the source code that will be used to compile the executable system has been created during the development phase and shipped with the source distribution, except a few system-dependent header files that are automatically generated by `configure`; most of these files contain simple macros and system-dependent constants. However, GCC has a much more complex and dynamic building process, where the software architecture undergoes a significant reshaping — some of the most important source code files in the GCC system are automatically generated, then compiled and linked to create the final compiler system.

One of the main design goals of GCC is to support multiple programming languages, hardware architectures, and operating systems. Not only does GCC support a wide range of languages and platforms "out of the box", the compiler suite has also been designed to allow easy expansion and customization by third-party developers. To achieve these goals, the architecture of GCC must be both flexible and configurable. To that end, the individual GCC compilers share many of the front-end and back-end components, such as parts of the compiler driver, the preprocessor, the RTL (an intermediate code format), the object code generator, and the optimizer. Support for multiple programming languages is achieved entirely within the downloaded source code of GCC; however, support for multiple CPU architectures is implemented largely by generating part of the source code at build-time, based on information gleaned about the target environment by the configuration process.

In GCC, the Register Transfer Language (RTL) is an intermediate representation used to represent the target system's code after parsing, similar to Java byte code. However, unlike Java bye code, RTL is hardware dependent. The specification of RTL and the portion of source code that operates on RTL are generated at build time, using machine description information and collected system parameters from `configure`. The main benefit of having a target-dependent RTL representation is that we can immediately generate the target machine language (assuming an infinite number of registers), but in a way that the compiler can understand and manipulate the emitted instructions. Hardware-dependent optimizations also operate on this intermediate format, and only valid instruction for the machine is generated as results of all passes of transformation, for RTL has built-in knowledge about target CPU architecture [10].

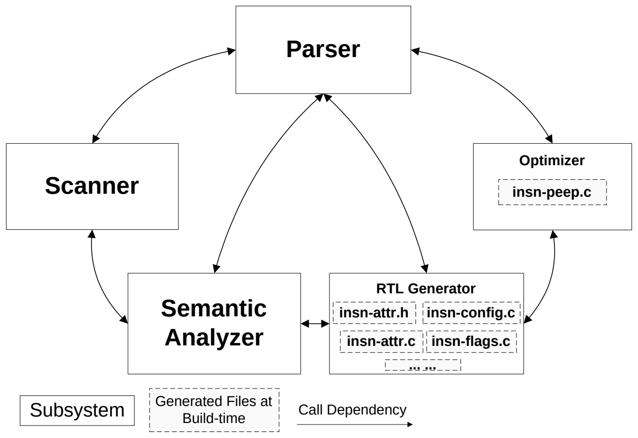

Figure 4. Code architecture view of the compiler-core subsystem in GCC.

Figure 4 shows a portion of the code architecture view of GCC 2.7.2.3 with "holes" (dashed boxes) that represent the "missing" source code files. The GCC system consists of five major subsystems: the driver, the preprocessor, the compiler-core, the code-generator and the utility libraries. Both the compiler-core subsystem and code-generator subsystem contain the RTL manipulation code that is missing from the distribution. The internal code architecture of compiler-core subsystem is illustrated in Fig. 4.

The "missing" files in the compiler-core subsystem are generated at build-time from code templates by source code generators.[3] The procedure is explained here and illustrated in Fig. 5 with a build architecture view diagram. In addition, Fig. 5 shows the relationship between build architecture view and code architecture view from above, and execution view from below.

1. First, the (build-time) source code generators are compiled; the source code for these generators are contained within files whose names begin with "`gen`". The result is a set of executable programs.

[3]The source code generators shipped with the GCC source should not be confused with the object code generator subsystem of GCC, which is a standard component of any compiler.

Filename	Role
`sparc.h`	Contains C macros that define the general attributes of the Sun SPARC architecture
`sparc.c`	Contains machine description supporting functions and macro expansion
`sparc.md`	Contains RTL expressions that define the instruction set (template). This file provides the input to programs that produce `insn*.h` and `insn*.c` files.

Table 1. GCC machine description files for Sun SPARC.

Filename	Created by	Description
`insn-attr.h`	`genattr`	Definition for any defined attributes and delay-definitions
`insn-attr.c`	`gen-attrtab`	Functions to access the attributes
`insn-codes.h`	`gencodes`	Definitions for named pattern

Table 2. Some GCC source files generated at build-time.

2. Next, these code generators are executed in sequence. They take machine description files for the target machine as input. The machine description files are picked by `configure`. Table 1 lists the machine description files for Sun SPARC architecture. The output is a collection of C source files, such as those listed in Table 2; these generated files have names that begin with "`insn`". These C files are used to fill the "holes" in the code view of compiler-core subsystem.

3. Finally, the source files to build a working GCC are all available. We now compile the code from the source distribution together with build-time generated code, and link them together to create the GCC compiler system.

Thus, the build-time architecture shows how the GCC system "fills in the gaps" of the code view of the shipped source code that were (intentionally) left by the GCC developers.

3.2 Build-Time Architecture of Perl

Perl version 5.6 [16, 8] is an interpreted high-level programming language, originally developed by Larry Wall. It is widely used to implement CGI programs, rapid prototypes, and system administration tasks. Like GCC, Perl is an open source system, which means that portability of the source is a paramount design goal. Also like GCC, Perl exhibits complex and interesting properties in its build-time architecture, which we now discuss.

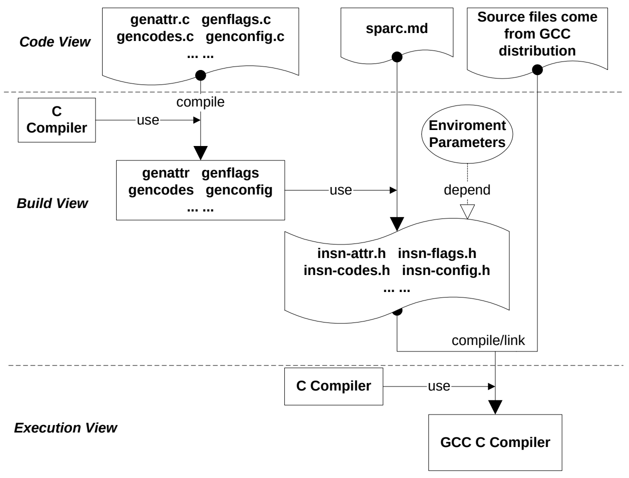

Figure 5. Build-time view of GCC (source code generation).

Filename	Description
`B.c`	Allow a Perl program to delve into its own innards. It is used to implement the "backends" of the Perl compiler.
`ByteLoader.c`	Used to load byte compiled Perl code. It uses the source filter mechanism to read the byte code and insert it into the compiled code at the appropriate point.
`DB_File.c`	Allows Perl programs to make use of the facilities provided by Berkeley DB.

Table 3. Some C files generated at build-time in Perl.

By comparing the content of Perl source directory before and after build, we found that there are 22 C source code files newly generated during the "`make`". The log file of `make` reveals that these C files originate from templates written in a special language called XS. The XS templates are then processed by an intermediate Perl interpreter `miniperl` with a translation script `xsubpp.pl`. The results of the processing are those new C source files mentioned above. Finally, the C files are compiled and linked by C compiler `gcc` as a part of Perl 5.6 run-time libraries.

Table 3 lists the filenames and descriptions of some of the C files that are generated at build-time; these particular files enable the Perl system to call Unix system libraries, Berkeley DB and Sys V IPC. It also enable Perl programs to directly manipulate the backend Perl run-time system. These C files are part of the Perl source code but they are generated at build-time. The templates for the C files are written in a special language called XS and they come with the Perl source distribution. We now explain the XS language and its relationship to Perl [16].

Assuming you want to implement a function in C or call a C system library from Perl, you need to write a special C library that can be either dynamically loaded by the Perl run-time system or statically linked into the Perl executable. This library acts as "glue" that links the C code with the Perl system. For example, when a C library function is called from the Perl program, the glue library first pulls arguments of the call from Perl's argument stack and converts the Perl values to the formats expected by the particular C function. Then, it calls the C function and finally transfers the return values of the C function back to Perl. To pass the result back to Perl, the glue library either puts them on the Perl stack or modifies the arguments supplied by Perl.

Of course, a programmer can craft a "glue" library directly in C, but this is very tedious and requires knowledge about the working mechanisms of Perl stack. Therefore, the Perl distribution provides a standard interface and mechanism to create glue libraries that is both flexible and easy to use. The interface is written in XS language that allows a programmer to describe the behaviours of the glue. The XS language describes the mapping between a Perl function and a C function. It can also define a wrapper Perl function that wrap around a C function. The XS compiler `xsubpp`, which comes with the Perl source distribution, compiles the XS file and generates the glue automatically. The output of `xsubpp` is the C source file that implements the glue library, as we can see in Table 3.

Now that we understand the procedure to create a Perl-C extension, we can investigate and create models for the build-time architecture of of Perl 5.6. The Perl source distribution includes 22 Perl-C extension templates written in XS language. These extensions are either part of the Perl run-time environment, or interface to important Unix/C libraries. The translation process to generate the Perl-C extension from the XS templates is operating system dependent. Therefore, these extensions are created at build-time, since the operating system environment parameters are only available after running `configure` on the build platform.

Similar to GCC, Perl also bootstraps itself. An intermediate Perl interpreter called `miniperl` is built first, then `miniperl` and `gcc` are used together to build the complete Perl interpreter and run-time system. The translation script that transform XS file to C file is also executed by this `miniperl`.

The build architecture view of Perl is shown in Fig. 6,

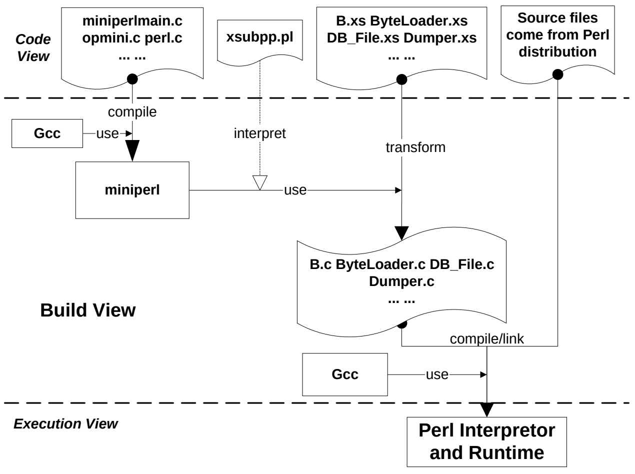
Code View
miniperlmain.c opmini.c perl.c ... ...
xsubpp.pl
B.xs ByteLoader.xs DB_File.xs Dumper.xs ... ...
Source files come from Perl distribution
compile
Gcc
use
interpret
transform
miniperl
use
B.c ByteLoader.c DB_File.c Dumper.c ... ...
Build View
compile/link
Gcc
use
Execution View
Perl Interpretor and Runtime

Figure 6. The build-time view of Perl 5.6.

where the build-time code generating and bootstrapping are clearly illustrated. At the top of Fig. 6, the code architecture view corresponds to the directory structure and organization of Perl source code. Components of the code architecture view shown here include the source files to build `miniperl`, the translation script from XS to C `xsubpp`, Perl-C extension templates `.xs` files and other C code to build the rest of Perl interpreter and run-time.

The first step of build creates an intermediate Perl interpreter `miniperl`, which provides a functionality subset of the full-scale Perl interpreter. Although `miniperl` is not as efficient and optimized as the final Perl interpreter `perl`, it is sufficient to accomplish the required tasks.

The second step is to create another important component of the build architecture: the C files that implement the "glue" libraries for system Perl-C extensions. The translation is carried out by the Perl interpreter `miniperl` and translator utility script `xsubpp`. The translator script takes `.xs` files as input, and outputs the C files, such as those listed in Table 3.

The final step is to compiler all of the C source files and linked them together to create the Perl interpreter and run-time libraries.

3.3 Build-Time Architecture of the JNI

The Java Native Interface (JNI) is a programming interface for Java. Using the JNI allows Java programs running within a Java virtual machine (JVM) to interact at run-time with applications and libraries written in other languages, such as C, C++ and assembly language. Similar to the Perl-C extension provided in Perl, JNI enables Java programs to take advantage pre-existing native applications, to exploit platform-dependent utilities, or to improve the performance

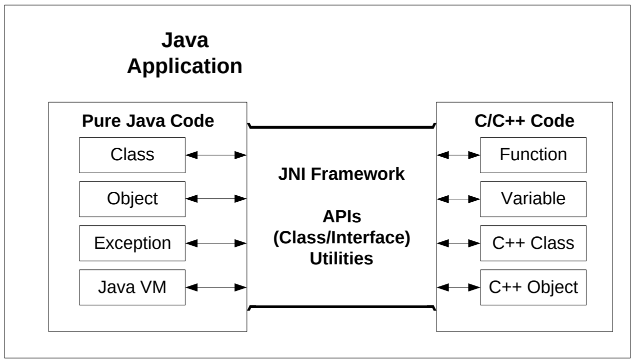

Figure 7. The JNI (Java Native Interface) framework.

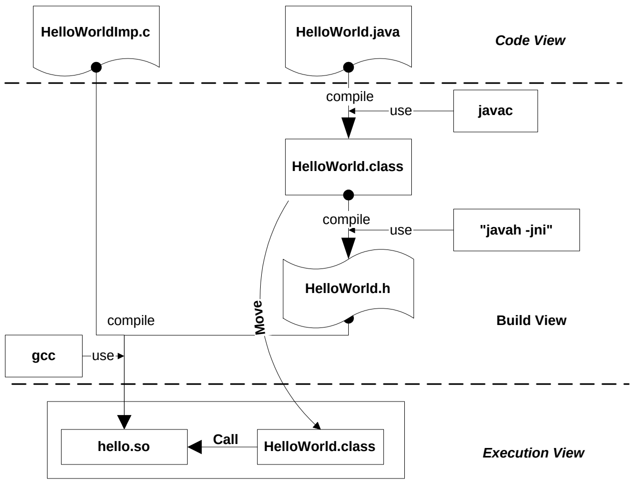

Figure 8. The build-time view of JNI.

of system bottlenecks [4].

Similar in spirit to the Perl XS interface, the JNI also provides a standard framework with a collection of Java APIs and tools. They work together to allow the programmer to “glue” Java application with native methods (typically, C or C++ library functions), as shown in Fig. 7.

A typical use of JNI involves several steps:

1. First at the development stage, we write Java code that declares only the signature of the native method. The native method can be called from any other part of the Java program as if it were a “normal” Java method. We also implement the native method in C or C++.

2. At build-time, we compile the Java class that declares the native method and all other Java classes including those who call the native method. Next, we use the JDK tool `javah` to generate the header file for the native method. Now we have the native signature for our C/C++ implementation. Then, we compile both the header file and the implementation C/C++ file into a shared library file.

3. Finally, we have the running application, where the Java code from the JVM can call the functions defined in the native library.

A build-time architectural view of using Java JNI is shown in Fig. 8 using our notation. This example shows a Java program that calls a C program that prints “Hello world” [4]. We can compare its to the the build-time view of Perl shown in Fig. 6.

4 Build-Time Views and Configuration/Build Management

In the previous section, we examined the build-time architectures of GCC, Perl and Java JNI system, and we created diagrams to model the architectural views. We now give an informal definition of our notation for visualizing build-time architectural views.

As shown in Fig. 9, we define four types of components and three types of connectors in our build-time architecture view:

Components: A box with a straight top edge and a curved bottom edge represents source code that has been written before the build and is provided within the original source distribution. A box with curved top and bottom edges represents source code that is dynamically generated at build-time. A box with straight edges represents a program executable or Java class file. An ellipse denotes environmental information, such as CPU and operating system details.

Connectors: For a compilation link, an arrow is drawn from the source code to the compilation target; the compiler used may be indicated as an association to the relation arrow. A common generalization of this connector is the interpreter/translator arrow, where a transformation is performed based on a provided script; in this case, the script is denoted by the addition of a hollow-headed arrow. The third type of link is a general build dependency, which is represented with a hollow-headed arrow.

Build-time architectural views can serve as high-level and visual documentation for software configuration and build management. As the build architectures of software systems have become more dynamic and complex, the importance of explicitly modelling build-time artifacts and their related activities increases.

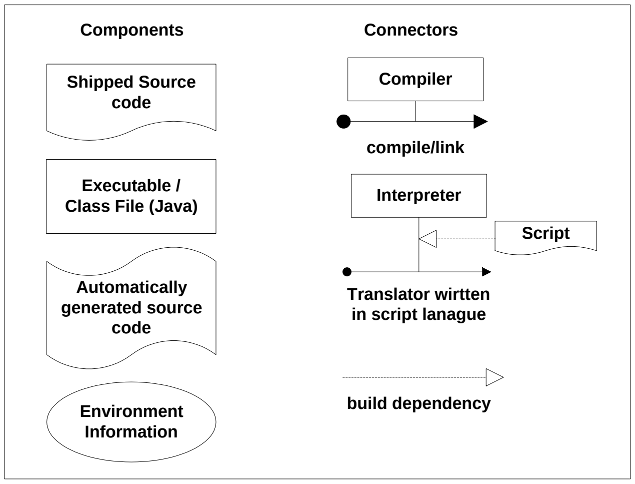

Figure 9. Notation of build-time architectural view.

One common reason to have a complex build-time architecture is to aid in the cross-platform construction problem, when software systems are required to build and run on a variety of hardware platforms and operating systems using the same source code distribution. The usual solution involves running a configuration tool, such as the Unix tool `configure`; such a tool queries the operating system for hardware and operating system details. This information is then used together with a set of provided `Makefile` templates to create customized build scripts that are tailored for the target environment. Often, this information is straightforward in nature, and entails the setting of various source code macros and compiler flags. However, in the case of GCC the data structures of the Register Transfer Language (RTL) and algorithms to manipulate them are fundamentally different for each CPU architecture. The task is far beyond the capability of `imake` and conditional compiling techniques. The solution employed by GCC is to develop a special program to behave as an automatic code generator. It takes architecture description files as input and emits appropriate algorithms and data structures as output. As a result, some fundamental parts of the source code for GCC are created during build-time. This unique design approach makes the build-time architecture of GCC very dynamic, and it is not well represented by static architectural views. Many interesting dynamic build architectures may be found within the domain of programming language compiler/interpreter systems. This is because the data structures and algorithms (especially algorithms for code generation and optimization) are highly dependent on the target CPU architecture and operating system.

It is important to point out that there are many software/computing systems that use various dynamic configuration techniques at run-time, such as COM/DCOM, CORBA, Enterprise Java Beans, and Java reflection. For example, DCOM and CORBA allow distributed applications to call a remote service without knowing its exact location or even the programming language in which it is implemented. However, these techniques are part of the execution architecture, where the distributed components have already been build and deployed. Their dynamic run-time architectures can be modelled by the "physical view" in the "4+1" model and the "execution architecture view" in the "four views" model.

5 The "Code Robot" Architectural Style

In their book on software architecture, Shaw and Garlan discussed several *architectural styles* that model recurring abstract patterns of high-level structure within software systems, such as "pipeline", "layered", "client-server" and "interpreter" [13]. In Kruchten's "4+1" paper, each of his architectural views indicates some representative architecture styles. For example, the "object-oriented" style is used in logical view, while the "pipe and filter" and "client-server" styles are applied in the process view [9].

We consider that the dynamic build-time behaviour of systems such as GCC and Perl defines a new architectural style that applies to build-time architectures; we call it the "code robot" architecture style. The idea is, if the behaviour of the software system depends heavily on the hardware architecture or operating system, the software designer must devise an effective and sophisticated strategy for customization of the system source code at build-time. For example, the strategy taken by the developers of GCC and Perl is to write a code generator, a "code robot", such as the `gen*.c` in GCC and `xsubpp` in Perl. In Perl, the system-dependent code is specified by a template written in XS language. In GCC, the rules of how to create them from hardware architecture description files are embedded in the code robot itself. Given a description of hardware architecture, the code robot knows how to generate corresponding system-dependent code. Figure 10 shows the "code robot" architecture style.

5.1 Code Robots and Open Source Software

Automatic source code generation is not new; it is a well known technique that is in wide use in both industry and by open source projects. For example, the code wizard in Microsoft's VisualStudio can create generic MFC skeleton code for a Win32 application, which includes standard windows, menus, dialog boxes, and shortcuts. Others examples include `lex`, `yacc`, and their relatives which generate scanners and parsers for compilers.

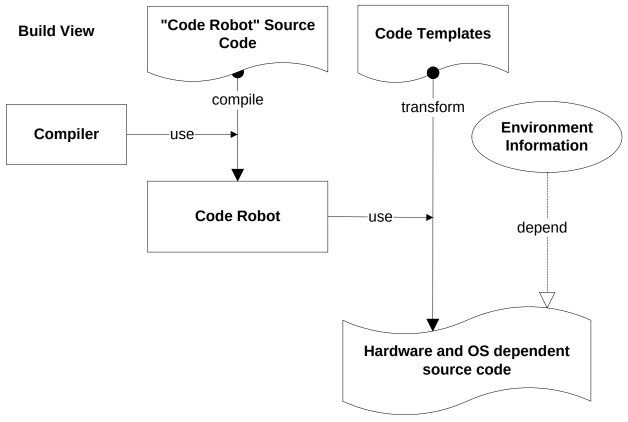

Figure 10. The "code robot" architecture style.

Automatic source code generation *during system building* is much less common. The reason seems to be that most commercial software systems are targetted for a relatively small number of possible hardware architectures and operating systems. Most companies do not ship the source code for their systems to clients; consequently, system building is performed prior to shipping, and the software company can choose its preferred platforms, compilers, subtools, and libraries. This means that system building can be managed in a straightforward manner, often by creating a product line for each target environment.

However, open source software systems are, in general, designed to be as portable as possible. Since the source code is available, the system is typically built by the user; this means that the source code distribution must be designed to buildable on a large number of possible platforms, using different compilers and subtools, and using different system libraries. Rather than create a separate source distribution for each set of alternatives, open source systems usually abstract the commonalities into a single distribution and rely on configuration tools to aid in building. This is why the code robot style is most commonly found within open source software systems.

6 Summary

In this paper, we identified a new aspect of software architecture: the build-time architectural view. We extended the popular "4+1" view model of Kruchten to capture the additional concerns that relate to the building of software systems. We explored the characteristics and the significance of software build-time architectures in three case studies: GCC, Perl, and Java's JNI. Through our case studies, we discussed how explicitly modelling build-time behaviours with these architectural views can aid developers in gaining a better understanding the software system itself, as well as in managing the build process in the software development/deployment cycle. Finally, we introduced the "code robot" architectural style to aid in modelling build-time views of systems, and discussed why it is most common in open source software systems.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Priciples, Techniques and Tools*. Addison Wesley, 1986.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. Addison Wesley, 1998.

[3] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. of the 21^{st} Intl. Conference on Software Engineering (ICSE-21)*, Los Angeles, CA, May 1999.

[4] M. Campione, K. Walrath, and A. Huml. The Java tutorial: A practical guide for programmers. Website. http://java.sun.com/docs/books/tutorial/java/.

[5] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proc. of 2000 Intl. Symposium on Constructing software engineering tools (CoSET 2000)*, Limerick, Ireland, June 2000.

[6] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 2000.

[7] http://www.gnu.org/software/gcc/. GCC homepage. Website.

[8] http://www.perl.com. The Perl homepage. Website.

[9] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(5), November 1995.

[10] H.-P. Nilsson. Porting GCC for dummies. Website. ftp://ftp.axis.se/pub/users/hp/pgccfd/.

[11] D. E. Perry. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[12] T. Pittman and J. Peters. *The Art of Compiler Design, Theory and Practice*. Prentice-Hall, 1992.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[14] The Perl mailing lists. Website. http://www.perl.org/support/mailing_lists.html.

[15] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architecture analysis and repair of open source software. In *Proc. of 2000 Intl. Workshop on Program Comprehension (IWPC'00)*, Limerick, Ireland, June 2000.

[16] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, 2000.