Representing Build-Time Software Architecture Views with UML

Qiang Tu and Michael W. Godfrey
Software Architecture Group (SWAG)
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{qtu, migod}@plg.uwaterloo.ca

Abstract

We have found that some classes of software systems exhibit interesting and complex build-time properties that are not explicitly address by existing models of software architecture. In this paper, we briefly explain the idea of *Build-Time Architectural Views*, and demonstrate how to model them with extended UML notations.

1. Architecture Views

Kruchten proposed the "4+1" view model to include Logical View, Process View, Implementation View, and Deployment View. Scenarios and use-cases show how views work together to satisfy user requirement [1,2]. Hofmeister et al. have defined a similar taxonomy, which we shall refer to as the "four views" model: Conceptual architecture view, module architecture view, code architecture view, and execution architecture view each addresses different engineering concerns [3,4].

Both the "4+1" view model and the "four views" model concern three basic types of views: Requirement Views (logical, process, scenarios, and conceptual), Development Views (implementation, module, and code), and Deployment Views (deployment and execution). However, we have noticed that many systems, especially Open Source Software systems such as GCC and PERL, exhibit interesting structural and behavioral properties that are apparent only at system configuration and build-time, and that these properties are not explicitly considered or appropriately modeled by either the "4+1" view or the "four views". Therefore, we now examine the idea of Build-Time Architectural Views [5] in more detail.

2. Build-Time Architecture Views

Once a large software system has been implemented in a particular programming language, it must be configured, compiled, and linked for a specific runtime environment before it can be system tested or deployed. For small software systems written for a single platform, the `make` utility and a simple `Makefile` is often sufficient to manage the system building. However, for systems that are large and complex, that run on a variety of hardware platforms, and that support multiple functional configurations, the task of build management is non-trivial. For such large and complex systems, build management is typically performed by dedicated employees (build engineers) using specialized build/configuration management systems. 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 procedure.

Build-time architecture views capture configuration and build-time properties that are present in the build management artifacts, such as build scripts, system source code, object files, binaries, library files, and build configurations. At the very least, they should model the compilation dependencies among compilation units, link dependencies among object/intermediate files, time-sequence configuration of the build procedure, and source code generated by build-time.

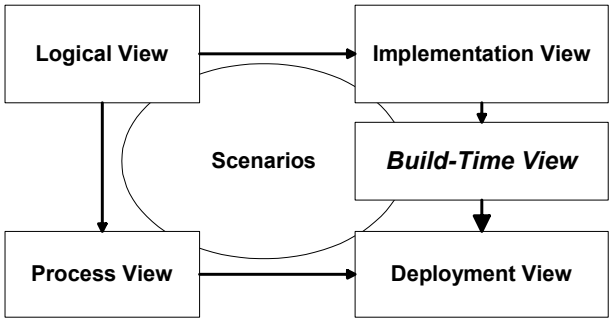

Figure 1. Build-Time View in "4+1" model

In Kruchten's "4+1" view model, *Implementation View* (previously called *development view*) models the static organization of the software in its development environment. It focuses mainly on how the source code of the software is structured and managed. *Deployment View* (previously called *physical view*) shows how the software elements are mapped onto the hardware execution environment. It captures the system topology, delivery, installation and communication aspects of the system. Our proposed *Build-Time View* should be positioned between the *Implementation View* and *Deployment View* as shown in figure 1. It captures the static building dependencies between source code entitles, as well as build-time behaviors and procedures when we actually build the software system on the target environment.

When the development phases is complete, the build engineers write the `Makefile` or other automatic build scripts and build the system from a collection of source code files in the *Implementation View* into a suite of intermediate object files, executables binary files and static/shard library files in the *Build-Time View*. Then the artifacts from *Build-Time View* are packaged, shipped, installed, and finally configured into runtime entities in the *Deployment View* that work together closely to deliver the functional and non-functional requirements demanded by the customer.

3. Build-Time Views in UML

Build-time Architecture Views can be modeled using two types of UML diagrams. The static configuration of build-time architecture components (such as source files, object files, executables, and environment information) and the dependencies between them are modeled by the UML component diagram. The meta-model shown in Figure 2 is based on the *Code Architecture View* in “four views” model, but it focuses solely on the build dependencies and configurations. It also extends the elements and semantics from the code architecture view.

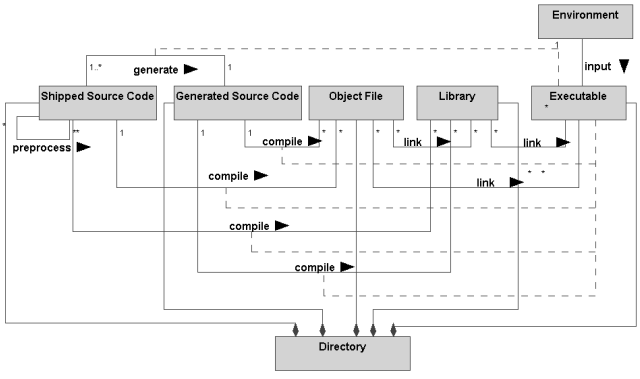

Figure 2. Meta-model for static build configuration view

One important extension to the original code architecture view meta-model is that for some relation links, we attach an association class/component to the relation. The association component explicitly specifies which compiler is used to compile source module A into object module B or which code generator is used to transform source module C into another source module D.

For example, GNU Compiler Collection (GCC) uses a multiple-pass compilation process, which is also known as "bootstrapping". During the "bootstrapping", three different GCC compilers are built at different time. This is a useful technique to build cross-compiler or update a system that only has an outdated compiler. We want to explicitly model bootstrapping in GCC with the build-time views to help programmers and build-engineers better understand the build-time nature of GCC software system, and also help them to comprehend the build script for easier maintenance.

Figure 3 demonstrates how to use the static build configuration view to model the bootstrap building of GCC version 2.7.2.3. The notations are based on UML component diagram, but we stereotype components to model various build-time entities, such as directory, source file, object file, executable file, and library file. We borrow the notation of association classes from UML class diagram to model which compiler is used for the compilation by attaching a component (compiler) to the compile link.

To model the dynamic behaviors of build-time architecture, we extend the UML sequence diagram as the modeling tool. It describes how different components defined in the static configuration view interact with each other at build-time, and most importantly, the time-sequence of building activities involving those components. Figure 4 shows the dynamic behaviors of GCC bootstrapping build view with a UML sequence diagram. Source code components are built by the compiler component to create a new instance of GCC compiler component. This process continues itself like a "snowball" until we have the final stage of GCC executables/libraries built.

With both the static configuration and dynamic behavior views, we have a clear picture that illustrates how the GCC bootstrap build is carried out at build-time: First, the existing C compiler is used to build the GCC driver, the C compiler and GCC static libraries from a subset of GCC source code. When the first build is completed, we have an intermediate GCC C compiler, which we would call "Stage 1 GCC". For the second build, we run the "Stage 1 GCC" over the same source code, with addition source code for a C++ compiler, a Object-C compiler and their

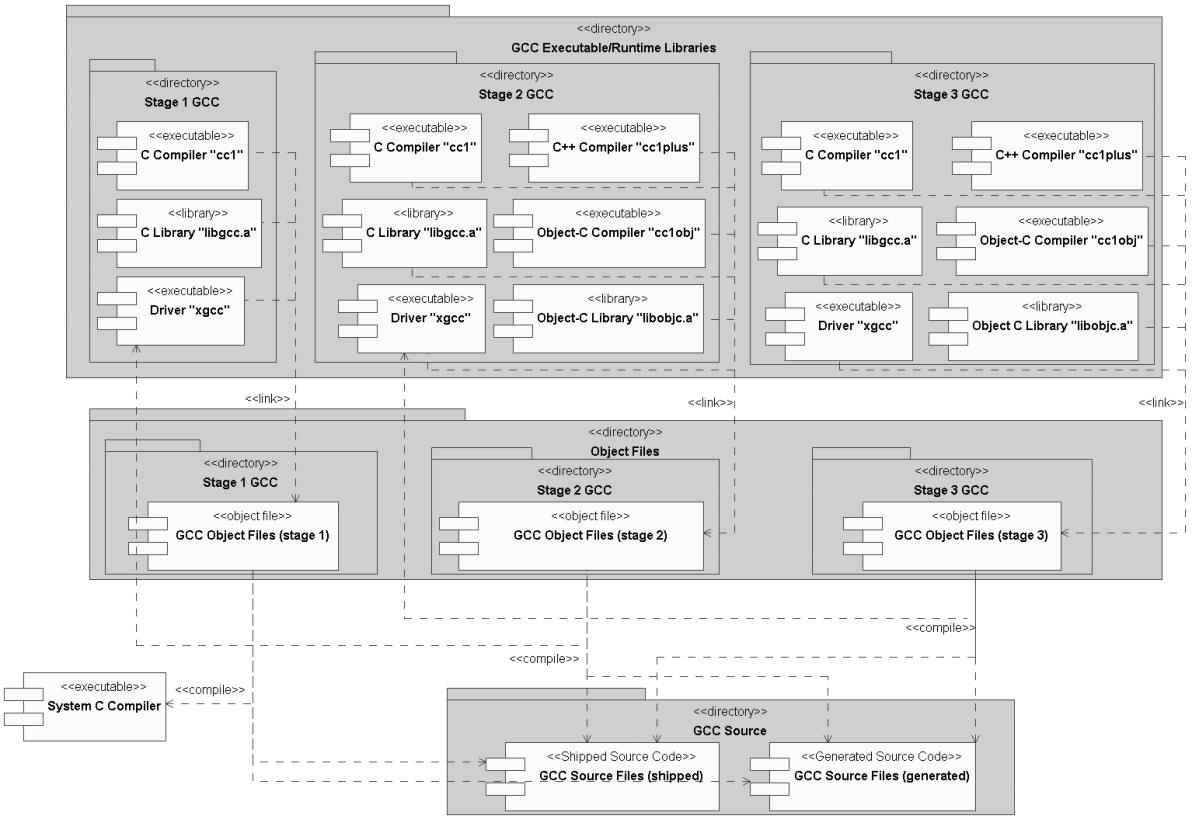

Figure 3. Static build configuration of GCC bootstrapping

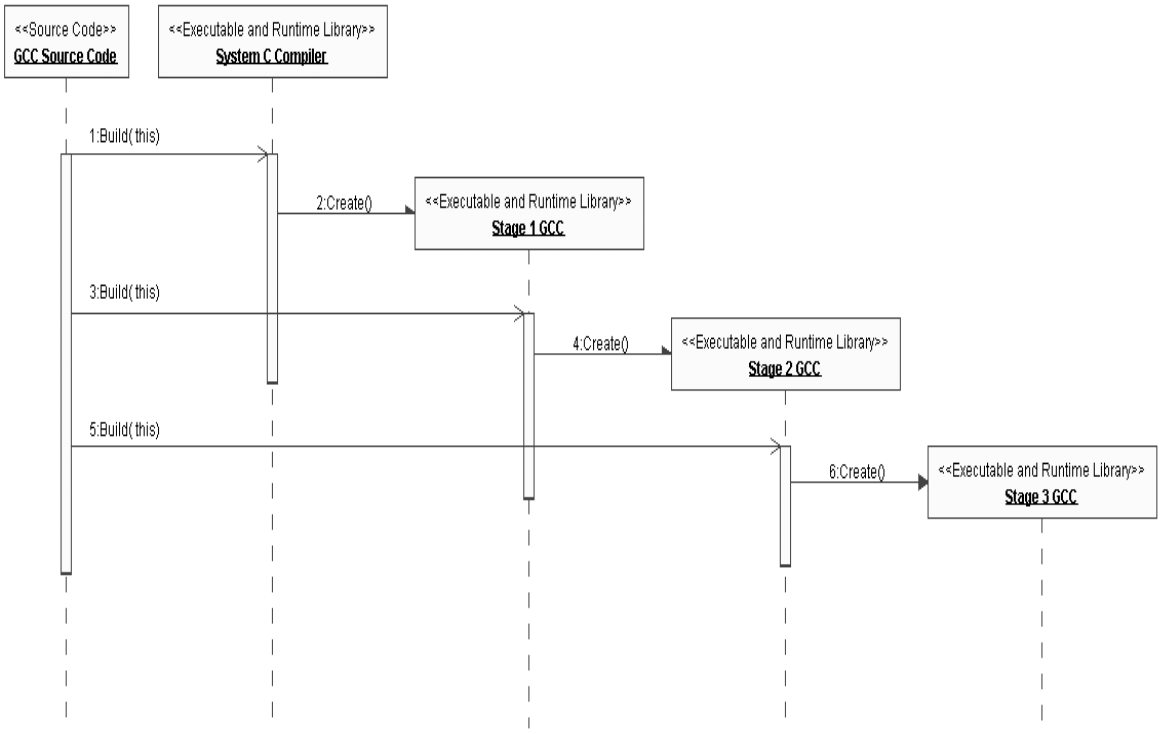

Figure 4. Dynamic build behavior of GCC bootstrapping

supporting libraries. When this build is done, all the functional components of GCC are built and integrated with a unified compiler driver `xgcc`. Then we use the "Stage 2 GCC" compiler to build the same source code as in previous build, and finally we have the "Stage 3" GCC. The final products will be deployed later to the target directory on target machine.

4. Another Build-Time View Example

In this section, we show another build-time view of GCC, and demonstrate how to use both the static and dynamic views to model building activities. Most software systems have their source code completed before the building starts. However, in GCC a significant number of core source files are generated during “build” by some customized code generators that take code template and target hardware/software environment as input and emit C source code. The generated source files are then compiled and linked with the rest of GCC source code to create the final GCC executables and runtime libraries.

GCC is an open source system, which means that portability of the source is a paramount design goal. This unique building behavior is part of the design effort to make GCC more flexible and configurable, in order to support multiple programming languages, hardware architectures, and operating systems. The portion of GCC source code that is generated at build-time implements support for multiple CPU architectures. It cannot be completely written before build because the source code depends on exact information about the target environment, and the deploy target is not possible to predict before build. The solution is to create a template for each supported CPU architecture, and implement a series of code generators. The code generators pick the right hardware template file at build-time as input, and emit corresponding source code that is hardware dependent.

Figure 5 and 6 illustrates this building procedure using static and dynamic build-time views. To make the static diagram more compact and focused, we omit all the object components from the diagram. First, the (build-time) source code generators are compiled; the source code for these code generators are contained with files whose names start with "`gen`". The result is a set of executables. Next, these code generators run in sequence. They take machine description files for the target machine as input. The output is a collection of C files, that all have names that start with "`insn`". Finally, the source files to build a working GCC system are all available. We now compile the code originally exist in the source directory together with those newly generated earlier in the build, and link them together to create a working GCC compiler system.

5. Conclusion

In this paper, we identified a new aspect of software architecture: the build-time architecture view. We extended the UML notion used in "four view" model to capture and describe the additional concerns that relate to the building of software systems. We demonstrate how to using UML static and dynamic diagrams to model the build configuration and behavior, and use the build-time view diagrams to explored the characteristics and significance of software build-time architectures in an Open Source Software system (GCC). With the case study, we discussed how explicitly modeling build-time behaviors with these architectural views can aid developers in gaining a better understanding the software system itself and some crucial decisions, as well as in managing the build process in the software development/deployment cycle.

Reference:

[1] P. Kruchten. *The 4+1 view model of architecture*. IEEE Software, 12(5):42--50, November 1995.

[2] G. Booch, *Software Architecture and the UML*. Slides, Rational Software.

[3] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Reading, MA: Addison Wesley Longman, 1999.

[4] C. Hofmeister, R. Nord, and D. Soni. *Describing software architecture with UML*. In P. Donohoe, editor, Proceedings of Working IFIP Conference on Software Architecture, pages 145--160. Kluwer Academic Publishers, February 1999.

[5] Q. Tu and M. Godfrey *The Build-Time Software Architecture View*, submitted to ICSM 2001

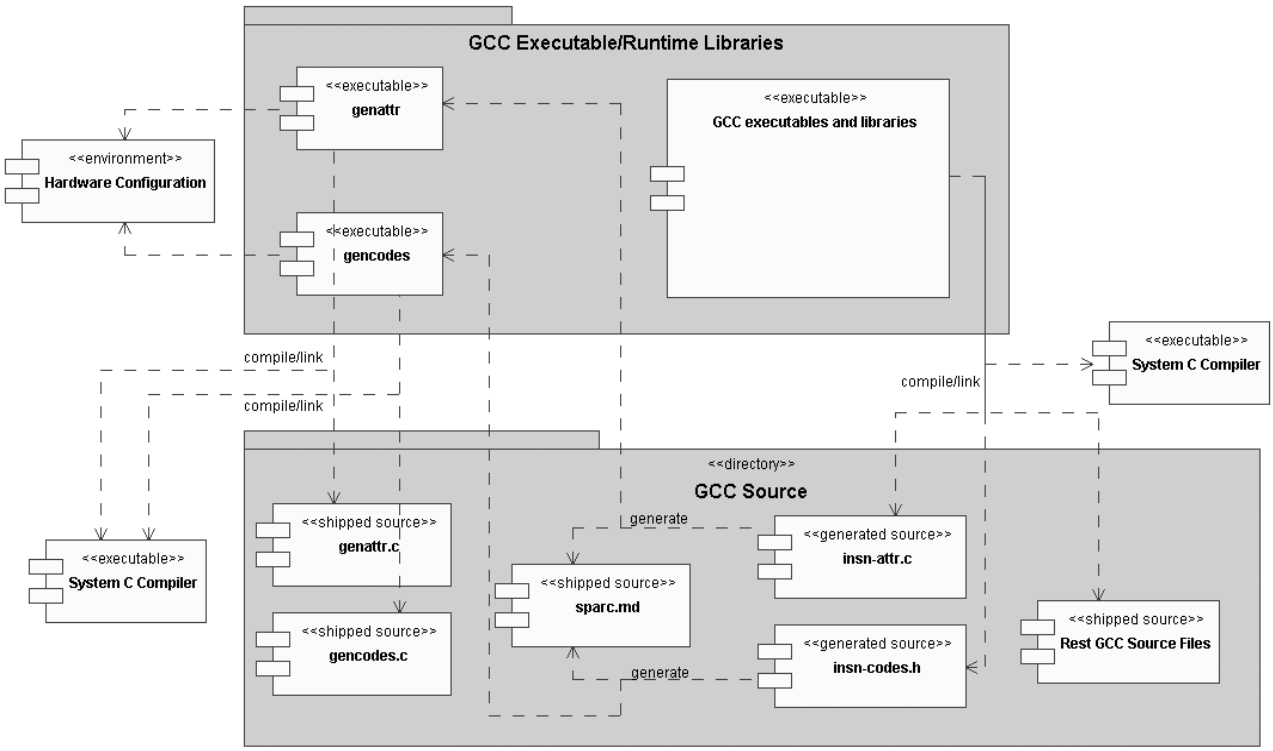

Figure 5. Static configuration view of GCC build-time source generation

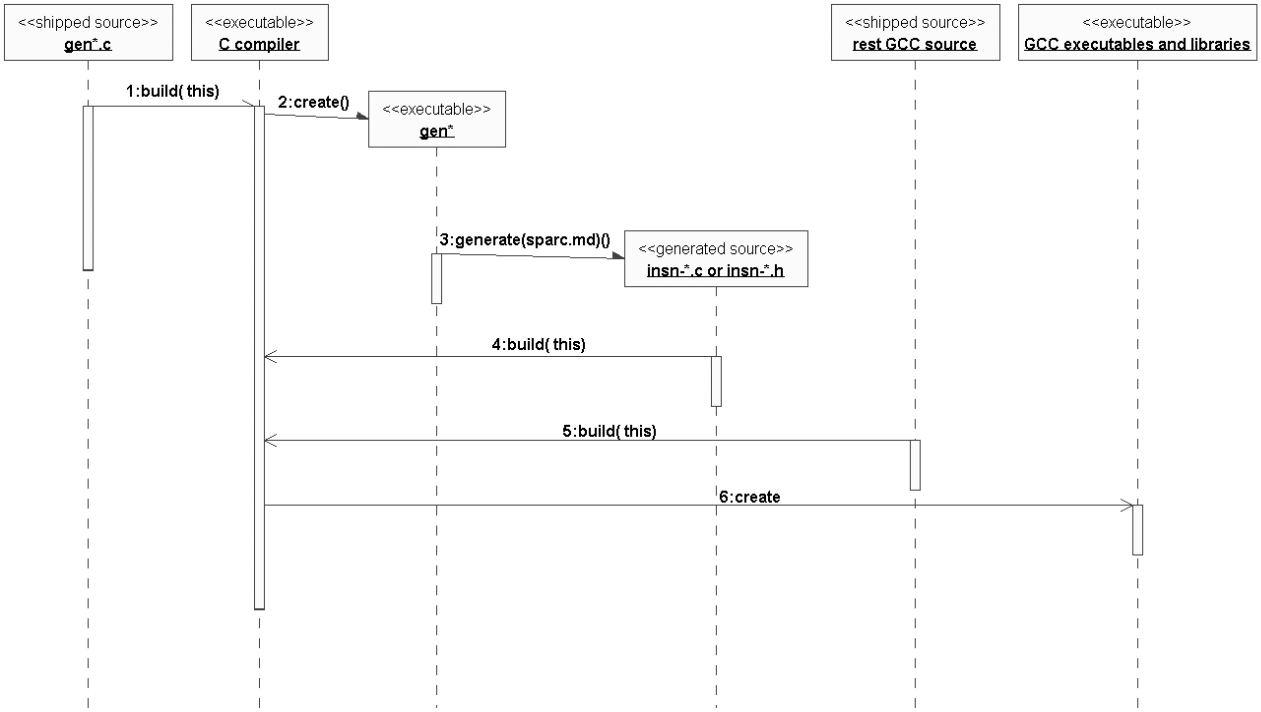

Figure 6. Dynamic behavior GCC build-time source generation