The Build / Comprehend Pipelines

Richard C. Holt, Michael W. Godfrey, Andrew J. Malton
Software Architecture Group (SWAG), University of Waterloo
{holt, migod, ajmalton}@uwaterloo.ca

Abstract

Large software systems often have complex subparts and complex build processes, and engage in subtle relationships with the underlying technologies from which they are designed and constructed. Most reverse engineering toolkits ignore the attributes and relationships of system construction; instead, they concentrate on static relationships among externally visible source code elements. This paper takes the position that the comprehension process for a large software system should mimic the system's build process.

1. Introduction

This paper takes the position that the comprehension process (and its supporting tools) for large software systems should be based on the steps for building the system. The main steps for building a system include (a) preprocessing, (b) compiling, and (c) linking, as well as various specialized steps, such as bootstrapping, source code generation, probing the targeted build environment, and code specialization [8]. To comprehend a large system, and to structure an appropriate reverse engineering process, one can take advantage of extant knowledge of these build steps, effectively creating a "comprehension pipeline" that shadows the build process. . Briefly, the first part of this comprehension process mimics preprocessing and compiling, and extracts a complete semantic image of each object module of the system. Each image consists of a graph, which includes an embedded AST (Abstract Syntax Tree) as well as edges and attributes recording types, declarations, dependencies, *etc*. The next step links the images of the object modules into an image (a large graph) of the executable code. To this image is added the architectural decomposition, dividing the system, recursively, into subsystems. This image is "shrunk" to manageable proportions by various abstractions, *e.g.,* removing function bodies, ignoring libraries, deleting built-in types, *etc.* Additionally, customized comprehension steps may be added to extract information about specialized build-time activities, such as source code generation or bootstrapping.

We now discuss the build and comprehension pipelines in more detail.

2. The Build Pipeline

The typical steps for building software are illustrated in Figure 1. It is tempting to view this process as a "black box"; however, this is not accurate. It is true that the transformation process is automatic once the source code has been completed; however, developers often use their knowledge of the build process to encode aspects of the design. These aspects are indiscernible by general-purpose source-based program understanding.

Two examples of this follow. First, early in the build process, source file inclusion establishes relationships between source entities (files) to indicate module dependency. Modules are not usually explicitly a language feature, but the questions "which source files belong to which modules" and "which modules does this module depend on" are essential to understanding at the architectural level.

Second, later in the build process, a linker (or even a dynamic loader!) has a search order for finding missing symbols. A programmer may use this search order to handle the configuration of software on different platforms, or to select the correct routine from one a collection of libraries with mutual name conflicts. In all cases a correct understanding of the software structure requires knowledge of the build process.

The steps of the build pipeline are well understood by developers, and it is our position that this understanding should be leveraged to help them comprehend a large software system. Developers understand that there are intermediate products emitted by each step in the pipeline, for example, object modules which are compiled versions of "compilation units".

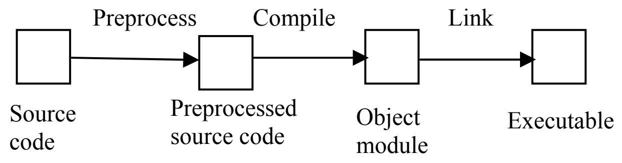

Figure 1. The build pipeline

Developers know that each of these products is a precise representation the semantics of (part of) the target system. Our goal is to capitalize on both this developer knowledge and on the structure of the build pipeline to provide a flexible and powerful approach to understanding large systems.

3. The Comprehend Pipeline: Front End

Figure 2 shows the front end of a reverse engineering pipeline that mimics the build pipeline from Figure 1. We will give a simplified description of this pipeline, and then will briefly explain how the actual pipeline works. The reverse engineering tools that the authors (and others) have developed, called SwagKit (Software Architecture Group Toolkit) [7], has this pipeline structure. It handles C and C++ programs and is based on the front end of the Gnu C/C++ compiler, but replaces the compiler's code generator with a program that emits the graph that is a semantic image of the object module. This modified compiler is called CPPX (C++ Extractor) [2]. These graphs are emitted in an intermediate ASCII language called TA [4] (or, optionally, in GXL [5]). The graphs for all of the target system's object modules are then linked into a complete semantic image of the executable system. This graph linker is written in the relational calculus language Grok [4,7]. We do not have space to go into details of various substeps in the comprehend pipeline, such as how repeated information represented in multiple object modules is merged following "raw link".

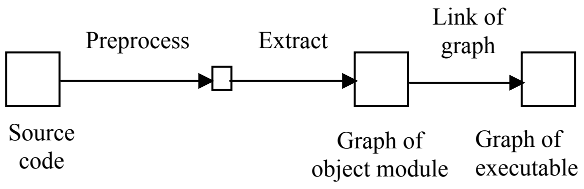

Figure 2. The comprehend pipeline: Front end

4. The Comprehend Pipeline: Back End

Figure 3 illustrates the back end of our reverse engineering pipeline. It produces useful views of the target system by means of two steps: the first step imposes a hierarchical or modular structure to the system. which is is a tree-based decomposition that breaks the system up into subsystems, which in turn are collected into subsystems, *etc*. with typically between 5 and 25 elements in each subsystem. This structure allows the system to be viewed and navigated one subsystem at a time.

These views, especially at the lower levels of the system, *e.g.,* within functions, would be overwhelmingly complex if we did not simplify them. This simplification is done in the "abstract" step of the back end. It uses a set of substeps written in Grok to eliminate and aggregate detail to the point at which each tree node with its children can be viewed comprehensively.

We have simplified our presentation of the comprehend pipeline by presenting it as if the steps were strictly sequential and as if there was no feedback from later steps to earlier steps. One of the essential re-orderings of the simplified pipeline we use is to abstract (or "shrink") before linking. That is, we "shrink-and-link", rather than "link-and-shrink" when possible to avoid dealing with massive graphs [1].

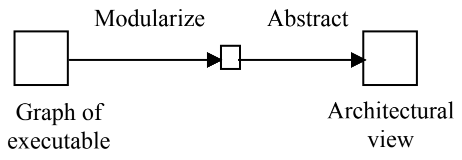

Figure 3. The comprehend pipeline: Back end

We have implicitly taken the position that the architecture of a large software system is fundamentally a view of its code. This position is reasonable because the developers necessarily need to understand the large scale structure of their code. This large scale structure is best thought of as the *concrete* architecture (or "development" view) rather than the *conceptual* (or "logical" view [6]).

5. The Comprehend Pipeline: Custom Steps

Some systems exhibit interesting architectural properties at build-time [8]. For example, the GNU Compiler Collection (GCC) first probes its target environment, then automatically generates some of its core data structures and algorithms based on the results, and finally compiles its resulting source tree three times. Systems that have such build-time properties require specialized techniques (such as build process instrumentation) to fully comprehend their architecture. Due to lack of space, we will not go into more detail..

6. Conclusions

Our position is that a reverse engineering approach that mimics the build process can be effective in revealing the target system's concrete architecture, and hence in helping us to comprehend the overall structure of the system.

References

[1] *Abstraction patterns for reverse engineering*, R.I. Bull, MMath thesis, University of Waterloo, 2002.
[2] The CPPX homepage, http://www.swag.uwaterloo.ca/cppx
[3] "Union Schemas as the Basis for a C++ Extractor", T.R. Dean, A.J. Malton, R.C. Holt, *Proc. of WCRE-01*, Stuttgart, Oct 2-5, 2001.
[4] "Structural Manipulations of Software Architecture using Tarski Relational Algebra", R. C. Holt, *Proc. of WCRE-98*, Oct 1998.
[5] "A Short Introduction to the GXL Software Exchange Format", R.C. Holt and A. Winter, *Proc. of WCRE-00*, Brisbane, Nov. 2000.
[6] "The 4+1 View Model of Architecture", P. Kruchten, *IEEE Software*, November 1995.
[7] The SwagKit homepage, http://www.swag.uwaterloo.ca/swagkit/
[8] "The Build-Time Software Architecture View", Q. Tu and M.W. Godfrey. *Proc. of ICSM-01,* Florence, Italy, November 2001.