

Increasing Software Development Productivity through Sophisticated Source Code Analysis

This article explains how to analyze and comprehend source code for reengineering and reuse, both for object-oriented programming languages like C++ or Java as well as for procedural languages (such as C). Wind River's source code analysis environment, SNIFF+, is taken as an example for a tool that offers comprehensive source code analysis and navigation capabilities.

INTRODUCTION

Today's development environments for embedded systems focus on aspects of development that are universally recognized as key requirements for embedded systems engineers. Such requirements include code editing, build and download functionality for specific target architectures and, with special focus, debugging the target system on various levels and tuning target system behavior with run-time analysis and profiling tools.

Surprisingly though, one important aspect of development is widely ignored: the substantial amount of time that developers spend searching for basic information buried in source code. Industry observers estimate that software developers spend up to 50 percent of their time searching for basic information buried in source code. And as applications grow in complexity, the time required to find, analyze and modify code grows proportionately to the total engineering effort.

As embedded computing has proliferated throughout the Internet economy, software applications - and the

projects undertaken to develop them - have become extremely complex. Today's embedded applications can run to millions of lines of code and originate from multiple teams working in a variety of development environments, languages and operating systems. Even with the most sophisticated integrated development environments (IDEs) and debugging tools, comprehending the structure, dependencies, and flow of complex applications requires higher orders of analytical sophistication.

Increasing complexity leads to constantly growing development lifecycles, introducing a severe need to reengineer software and to reuse existing software components. Tremendous amounts of legacy code and high staff turnover in today's information technology (IT) industry force developers to work with code that they have not written themselves or have never even seen before. The growing use of object-oriented programming techniques introduces additional complexity to developers accustomed to program procedural languages. These factors result in significantly increased time required to understand the structure and interdependencies of large code bases, representing a severe limitation to today's software development productivity. The solution to this dilemma can be found in powerful source code analysis tools, such as Wind River's SNIFF+ source code analysis environment.

ANALYZING SOURCE CODE

Comprehending source code involves understanding what an identifier stands for, what type it is, its properties and how it interrelates with other parts of the system. At this level of analysis, code statements such as expressions, loops or conditional execution statements are typically less interesting and dependent on runtime behavior. Therefore, structural and static source code analysis focus on finding identifiers, analyzing their types and structure as well as exploring and visualizing their interrelations and dependencies with other identifiers.

QUICK NAVIGATION

Complex directory structures and wordy file names in large applications can make navigation a nightmare. Even basic navigational tasks such as finding code

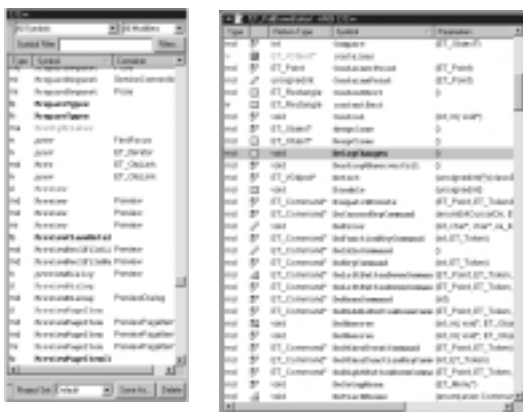


Figure 1 (left). The Symbol Browser provides an overview of the definition of symbols in the source code. The symbols can be filtered according to symbol type and other criteria.

Figure 2 (right). SNIFF+'s Class Browser gives in-depth information about all class members, their type and visibility, whether public, protected or private. It makes it easy to quickly understand class instantiations and how to reuse and inherit class members from base classes.

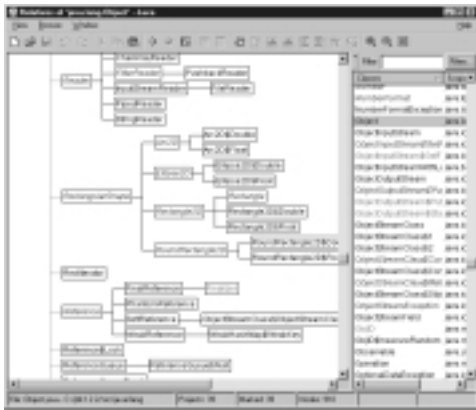


Figure 3. The Hierarchy Browser helps to understand complex C++ and Java class inheritance trees. It shows how class families relate throughout the entire system, where virtual methods are defined and implemented.

that is implementing certain functionality can be extremely cumbersome. The ability to quickly find identifiers with specific names, types or attributes is a simple requirement of source code analysis tools. But surprisingly, this requirement is often ignored. When working with large code bases, it is extremely helpful to have powerful semantic filtering. This enables the user to search for all static functions returning "void" with a certain name prefix or signature in one task. For larger projects with several thousand files, the number of identifiers that need to be analyzed can quickly grow into the millions - requiring sophisticated technical solutions and user interfaces to keep this amount of information immediately accessible and manageable to the user.

COMPREHENDING DESIGN STRUCTURES

In general, analyzing and comprehending the design structure of a software system is very complex, especially for non-object-oriented languages that lack a general design element. For object-oriented programming techniques, it is easier to comprehend the struc-

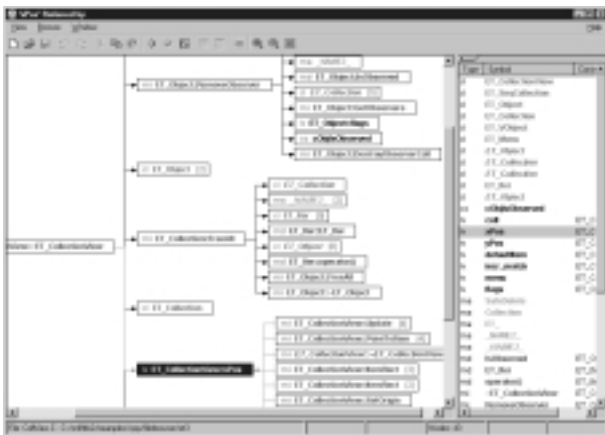


Figure 4. The Cross Referencer visualizes how objects are used throughout the entire system and how they depend on each other. Discovering the usage of methods and finding where a variable is read or written are fast and easy with SNIFF+.

ture because the classes and their inheritance graphs provide a first insight into the structure of a C++ or Java software system. This is because they reflect conceptual aspects from the pure definition of functional behavior, as interface or abstract class, up to several implementation layers for different purposes.

Ideally, a software analysis tool will visualize this structure by analyzing interface and abstract class definitions and, in this context, the classes that are actually implementing the functionality on class level. It will accomplish this by marking interfaces and abstract classes, as well as on the class member level, by visualizing definers of virtual members and those classes implementing or already overriding virtual methods.

To understand the functionality implemented in a class, the class interfaces need to be analyzed including the members inherited from base classes and their visibility. With semantic filters for visibility types, such as those found in the widely-used SNIFF+ product from Wind River, an immediate overview of the interface of a class to a derived class (reuse) or to the outside world can be obtained. This information is collected and shown in a single view, allowing the user to quickly understand how to use or reuse existing classes. With most standard tools, all class interfaces need to be reviewed and analyzed separately which proves to be a very complex and time consuming task for larger class hierarchies.

STUDYING INTERDEPENDENCIES

Having an overview of structuring elements does not yet explain how they interact and depend on each other. But understanding these relationships is key to any reengineering activity. Assuming a developer needs to change the signature of a function, the definition of a type or a class interface, then analyzing the impact of this change to a millions of lines of code system without sophisticated tool support is extremely time consuming. Having a robust analysis tool like SNIFF+, which is able to compute all occurrences of calls to a function or method and to find all accesses to a global variable, is a tremendous advantage. In addition, developers need to have a complete picture of all interdependencies in order to visualize how symbols are used as parameters or as components in classes and complex data types.

File level dependencies introduced by C / C++ pre-processing or Java import mechanisms often result in heavy maintenance efforts because a change in a header file can impact the entire software project. Visualizing these dependencies helps to detect unnecessary includes, which not only lead to longer build times but also to what may be unexpected changes after modifying a header file occurs.

AUTOMATING TIME CONSUMING CHORES - GLOBAL CODE CHANGES

Having exact information about all interdependencies allows the developer to go beyond visualization only. Once all areas impacted by a code change are identified, a global replace mechanism can automate what otherwise would become a cumbersome chore:

changing all effected locations. SNIFF+ allows the user to perform global code changes, based on cross-reference information, within a matter of seconds. Comparatively, the same task performed manually could require hours to complete. Clearly, not every change allows the impacted source code to be modified with straightforward replace techniques. But quite often this is possible and thus, any manual work performed to this end is largely wasted time.

INTEGRATION - KEY PRODUCTIVITY ASPECT

As previously explained, many different aspects of a system need to be analyzed in order to provide the user with comprehensive information. But to boost productivity substantially, a seamless integration of all capabilities is required. Conceptually, the various code analysis tools need to be integrated in a way that reflects the "natural" strategy a developer would use to gather information about unknown code. Let's assume you need to change a specific behavior in a C++ or Java software project. First, all classes must be found which may implement the functionality. Then an overview to their inheritance structure is required to see also all superclasses where the functionality may be implemented. All class interfaces then search for methods implementing the behavior - ideally by directly being pointed to implementations in the full class hierarchy. Once the method is found, cross reference information is required to understand how the functionality is implemented and all calls to this method need to be checked for potential side-effects. This simple example shows how quickly all different aspects of code analysis are involved and that tight integration is the key to boosting productivity.

FLEXIBILITY AND ANALYSIS TECHNOLOGY

Many standard IDEs use compiler output information for code analysis features. This appears to be a logical idea since a compiler is aware of all the details and interdependencies of the source code. However, in practice this approach has several pitfalls. Let's imagine you have to port a huge code base to a new target architecture. It is very likely that you will not be able to compile the code with the new compiler. So precisely when help is urgently needed, this technology does not work. Or let's assume that you need to understand the interface of a complex application program interface (API) and all you have are the C or C++ header files. No compiler-based code analysis tool will be able to provide meaningful information since there is no code to be compiled.

Due to these dilemmas, SNIFF+ uses its own parsing technology which extracts the information required for code analysis. It is fault tolerant and flexible enough to work with erroneous or even incomplete code. Obviously the speed of such specialized technology can be much higher, allowing you to immediately update the tools after file saves. This is another important productivity aspect since unnecessary delays for updating the information can be avoided.



Figure 5. The PowerChange tool automates global find and replace tasks. In seconds PowerChange performs complex changes in large systems and automates error-prone and non-productive tasks. It uses fast pattern and symbol based search and replace mechanisms to easily perform any kind of modification.

SUMMARY

Sophisticated code analysis tools can boost developer productivity dramatically. But their potential to shorten time to market continues to be widely ignored in an industry where time to market is the crown jewel of success. To achieve productivity improvements, code analysis environments need to provide information about all relevant aspects. These include fast navigation and information filtering, an overview of structural information and the ability to overview functional interfaces (mainly in object-oriented code). But most importantly, the code analysis tools should be able to visualize all kinds of interdependencies - identifiers used as class components or function parameters, read and write access to global variables and function and method call trees.

All aspects need to be tightly integrated to reflect the "natural methodology" of how developers would normally attempt to gather information about unknown code. To avoid unnecessary delays, specialized parsing technology can be used when huge amounts of code need to be reworked, are partially incomplete or not at all compilable. In these instances, tools like Wind River's SNIFF+ source code analysis environment become a software developer's best friend! ■

Stefan Hager is a product marketing manager for Wind River with special focus on source code analysis development tools, namely SNIFF+. Stefan joined TakeFive Software (Salzburg / Austria) in 1995 as Tech Consultant / FAE and has worked for Product Marketing since January 1997. He has been with Wind River since the company's merger with TakeFive in Feb. 2000. He holds a Diploma degree (Dipl. Inf.) in computer science from the Technical University Munich.