Build Management for AspectC++*

Andreas Gal
gal@cs.fau.de

Olaf Spinczyk
os@cs.fau.de

University of Erlangen-Nürnberg
Martensstrasse 1
91058 Erlangen, Germany

September 17th, 2002

Abstract

In large software systems the build process often takes a very considerable amount of time. When some source code files are modified, partial rebuilding of only the affected system parts is desired. For many years tools have existed, which can automatically detect dependencies between source code files and perform partial rebuilding. However, no common tool is currently able to deal efficiently with the crosscutting nature of aspects in aspect-oriented programming (AOP). As a result, changes to a single system aspect often trigger a full system build process. In this position paper we present our experiences with using AspectC++ to build large software systems. We present an approach to speed up the rebuild process by specifying aspect and component code dependencies in a declarative manner.

1 Motivation

By its very nature, aspect-oriented programming (AOP) is often applied in large software systems. The authors of this paper are using AOP for the implementation of PURE [2], a family of operating systems, PUMA [6], a C++ parser and source code manipulator, and AspectC++ [5], a general purpose aspect language implemented using PUMA. AspectC++ is a preprocessor for regular C++ compilers and emits standard C++ code after performing the aspect weaving. Both packages are written in C++ and AspectC++ is used as implementation language for the aspect code. In total, PUMA and AspectC++ consist of over 80,000 lines of code.

When we originally started to develop PUMA (and later AspectC++), we had no aspect-oriented programming language for C++ at our disposal. Thus,

*This work was partially supported by the German Research Council (DFG) under grant no. SCHR 603/2.

when we wrote the original code we could not exploit AOP. After AspectC++ was able to bootstrap itself, we started to modularize certain aspects of PUMA and AspectC++. We steadily replaced code scattered throughout the project with aspect code.

While this greatly improved the overall maintainability, the development process itself became painfully slow. Before we used AspectC++ for the development, our build utility GNU *make* [4] was able to quickly rebuild the system after a small change in a particular part of the system. It does so by analyzing the include dependencies of the C++ source code files. When invoking *make* after a change in the source code, the affected source code file and all files including that particular file are recompiled.

As aspects come into play, this approach of include file dependency analysis does not work anymore. Aspects are not limited to operate within certain source code file boundaries. Instead they can affect code throughout the whole project, wherever the join points of the aspect are located. To ensure a proper rebuilding of all possibly affected system parts, the whole system has to be recompiled after any changes in the aspect code.

One might be tempted to assume, that this will not dramatically affect the daily system development as long as component code represents the majority of source code in a given system and thus the majority of changes will be performed in the component code. However, this assumption is not correct. While the specification of a particular join point set is given within the aspect code, the concrete content of a join point set is mainly determined by the component code. For example, an aspect A my choose to give advice to all base classes of a certain class x. Lets assume x has two base classes y and z. In this scenario the aspect A dictates **what** the aspect weaver is performing in y and z, but the component code in x is determining **where** this manipulation takes place (y and x). If we decide to remove the base class y from x, we will be forced to recompile all three classes as the aspect A obviously does not apply to y any more. This is quite "unexpected" to the programmer, as he did neither change the base classes themselves nor the aspect A.

2 Dependency Analysis

Detecting such dependencies between component code and aspect code is extremely difficult. In pure functional C++ code the dependencies are simple to analyze: the *make* tool simply tracks all files included by the source code file in question. This analysis usually takes only a few seconds even in large projects.

Aspects, however, do not use simple *#include* statements to specify dependencies to other source code modules. Instead, complex expressions can be used to specify join points. To evaluate these expressions, explicit global semantic knowledge of the whole project is required. Thus, the source code of the whole project has to be parsed and semantically analyzed to do the dependency analysis. However, this is already the lion's share of the execution time of the AspectC++ compiler. The actual aspect weaving process is basically for free.

Thus, we would not gain much from the dependency analysis as far as speeding up the AspectC++ phase.

The subsequent compilation phase, however, could benefit from such a dependency analysis. As we mentioned earlier, AspectC++ is emitting standard C++ code, which has to be compiled with a regular C++ compiler to binary code. While the aspect weaving itself does not cost much, if we regenerate all the "intermediate" C++ code emitted by AspectC++, all of the code has to be compiled again to binary code, which takes a considerable amount of time.

A simple optimization, which can be applied here, is the use of compilercache [3] or ccache [1]. Both programs cache the compilation result of a particular input file instead of re-running the actual compiler every time. As in the average case the vast majority of the "intermediate" C++ code files do not change between builds, the compilation phase is dramatically accelerated.

3 Declarative Approach

While the system developed up to this point already greatly reduces the partial rebuilding time, it is still quite unsuitable for a number of scenarios. Besides "global" aspects like synchronization and distribution one also often encounters in aspect-oriented programming "local" implementation aspects. Just as "global" aspects these aspects do modularize code, but in a much smaller scope. For example, in PUMA we use aspect code to implement certain C++ language dialects (GNU C++, Visual C++). We chose to modularize this code with an aspect, because otherwise the parser code would be cluttered with dialect specific code. By design we know that these aspects only affect a few classes in the parser subsystem. However, any time we change a single line in this aspect programs or the parser component code, the whole 80,000 lines of PUMA and AspectC++ have to be reread and reanalyzed by AspectC++, which can take minutes.

Obviously the crosscutting nature of aspects does not allow an efficient automatic detection of component and aspect-code dependencies as it is possible for pure functional C++ code. Instead, we propose to use a declarative approach to explicitly specify the scope of aspects on the source file level. To prevent additional overhead for "global" aspects, where the scope should not be restricted, we understand these declarations as hints to the aspect compiler. If no such hint is specified, the aspect is assumed to (potentially) affect all source files in the system.

In AspectC++, the scope hints are given using a *#pragma* statement in the aspect code (Figure 1). We understand that the *#pragma* statement is very controversially discussed in the C++ community. The alternatives would have been hiding the hint inside C++ comments or extending the AspectC++ syntax. We disliked the idea with the C++ comments because of the possible collision with regular comments. Extending the AspectC++ was not an option either as we understand this approach as a build process "hint" not worth complicating the already quite complex C++/AspectC++ language.

```
aspect GNU_Extensions {
  #pragma weave "./CParser/*.*"
  ...
};
```

```
aspect VisualC_Extensions {
  #pragma weave "./CParser/*.*"
  ...
};
```

Figure 1: Giving scope hints in AspectC++

4 Conclusions

In this paper we motivated the importance of a quick system rebuilds for the development process and discussed the problems AOP causes in this context. We highlighted why traditional build management approaches fail to deal with the crosscutting nature of aspects in AOP and we presented a declarative hint mechanism to speed up the recompilation process by explicitly limiting the scope of aspect code to certain source files. While we do not understand the current mechanism to be the ultimate and final solution to this problem, we have reasoned that there is basically no alternative to this or similar declarative approaches as an automatic dependency analysis is too costly.

As far as future work is concerned, we are interested in extending the declarative description of aspects beyond the plain source file dependencies. We are confident that such declarative descriptions can lead to a better understanding and verifiability of component and aspect-code interaction as well as aspect code behavior in general.

References

[1] Andrew Tridgell. *The ccache Homepage*, 2002. http://ccache.samba.org/.

[2] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.

[3] Erik Thiele. *The compilercache Homepage*, 2002. http://www.erikyyy.de/compilercache/.

[4] Free Software Foundation. *The GNU Make Homepage*, 2002. http://www.gnu.org/software/make/make.html.

[5] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, Sydney, Australia, Feb. 2002.

[6] M. Urban. The PUMA User's Manual, 2000. http://ivs.cs.uni-magdeburg.de/~puma.