Bridging AOP to SMP:Turning GCC into a metalanguage preprocessor

Tiago Stein D'Agostini

Antônio Augusto Fröhlich

Federal University of Santa Catarina
Laboratory for Software/Hardware Integration
Florianópolis - Brazil

tiago,guto@lisha.ufsc.br

ABSTRACT

This article presents an strategy to combine important software engineering techniques, *Static Metaprogramming* (SMP) and *generic Programming* (GP) with *Aspect Oriented Programming* (AOP). These rely on specific language tools that, today, cannot be deployed in conjunction, thus imposing limitations on the software development process. Our strategy consists in adapting the C++ compiler to act as a SMP preprocessor. This preprocessor is able to parse the input program, execute eventual metaprograms, and output the resulting single-level program for further processing by an aspect weaver.

Categories and Subject Descriptors

Software Engineering [**Tools**]: [Aspect Oriented Programming, Static Metaprogramming]

Keywords

Aspects, Metaprogramming

1. INTRODUCTION

This paper focus on two software engineering techniques, weaver based *AOP*[2] and *SMP* [3] and in the interaction of these techniques. Today we have no tool capable of supporting AOP , SMP techniques at the same time: ASPECT-J is reaching maturity, but JAVA does not support SMP; C++ has full support to SMP and GP, but ASPECT-C++ [1]is restrict when the subject is weaving `templates`. For this reason, both techniques are seldom deployed together in the same project, thus restricting the level of modularity achieved.

This paper describes an approach to combine SMP (and consequently GP) and AOP techniques in the context of the C++ programming language, and *AspectC++* AOP language. This technique can also be used as a tool on bring *SMP* technique to environments lacking full featured C++ compilers.

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA
2005 ACM 1-58113-964-0/05/0003?5.00..

2. COMBINING AOP AND SMP

There are basically two alternatives to deliver the tool support needed to combine AOP with SMP in the context of C++: the first would be to include a metaprogram execution environment inside the aspect weaver, this would result including a complex part of a C++ compiler in the weaver. This alternative would incur in the implementaion of a complex feature, when it is already implemented somewhere else. The second alternative would be to deliver a metaprogram preprocessor to execute eventual metaprograms in advance of the weaving process, what would in fact provide a separation of the *SMP* and *GP* processing from the main compiling process. This alternative may bennefit from implementations of template handling in common compilers.

We chose the alternative of executing the metaprogram in advance, releasing the weaver from most of these tasks, leaving template instantiation as well as all other tasks concerning metaprogram execution, to be done by the *metaprogram preprocessor* tool tha wil be explained in the following section.

One point in favor of an external metaprogram preprocessor is the handling of join points targeting static metaprograms themselves. Since the preprocessor would be executed in advance of the weaver, metaprograms are no longer part of a program as it reaches the weaver, remaining only their results. This is important since the introduction of any code that cannot be solved at compile time inside of a metaprogram ,would invalidate it. It is a risky operation to weave traditional aspects when their joinpoints may match with metaprogram code. From this point of view, restricting the application of aspects only to metaprogram results is perfectly reasonable. Notice that this restriction does not apply to *Generic Programming*[4] , since the template classes are in the same language level as the aspects usually are targeting.

3. METAPROGRAM PREPROCESSOR

We developed a tool using GNU G++ 3.2 compiler, mainly because of its reliable capability of handling *SMP* and its

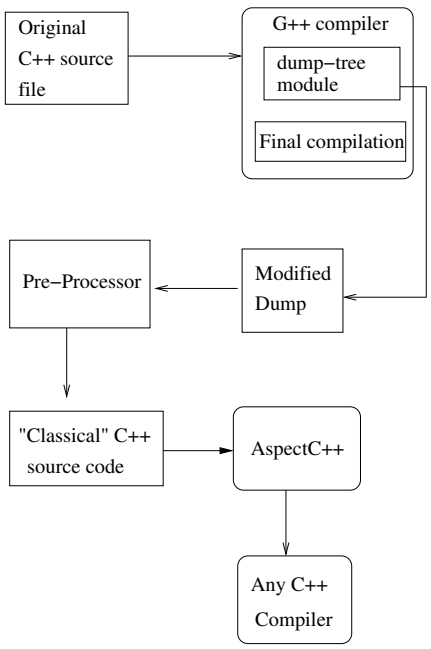

Figure 1: Overview

open source characteristic necessary for the implemented modifications. Our tool analyze data from a modified dump, originally provided by -FDUMP-TREE-INLINE option, although, the compiler was modified so that important information that were omitted on original dump system became available to the reassembling tool. All modifications applyed in to the compiler do not affect compilation process, only its dump system. The preprocessor can reconstruct the compiled program by using the program tree on a stage where all *SMP* and template code has been solved. The reassembling tool reads a parse tree for each function, and rebuilds the translation unit, replacing template classes and functions with new classes and functions corresponding to its instantiations. All static metaprograms are replaced by their results only. The resulting code is simple plain C++ and almost any modern compiler can handle it.

Issues arise with creation of template instances, which were not present in the original program. When new code is inserted in the program, there is a special concern about where to insert this code so that no undesirable side-effects are created. The problem is that templates can be instantiated with arguments whose types are declared somewhere else. For example, a parametrized class A<T> can be instantiated with any type argument that satisfies the constraints eventually defined by its methods. Considering that type B satisfy eventual constraints, class A<B> can be instantiated wherever type B is known. If we insert the declaration of A<B>—and any other eventual instances—at the point where the template was declared in the original program, it might happen that type B was not yet declared. This would invalidate the output program, preventing it even from a successful compilation.

An alternative would be to insert new classes corresponding to parametrized class instances just before the corresponding objects are declared. At this point, types eventually used as arguments to the parametrized class must have already been declared. However, a parametrized class instantiated in different scopes with the same argument set would incur in multiple declarations of the same type. Therefore, the solution is to insert the new classes in a point where all the types used as parameters of the template are already declared and inside a scope that can be reached by all instantiations of this template in the current translation unit. This requirements are accomplished in the outermost possible scope around the declarations where all types used as parameters of the template are already declared.

The reconstructed code can than be used as target of *AspectC++* or any other tool that cannot cooperate with *SMP* code, also the resultant code is less rich in complex features so that it may be compatible with less features compilers that would not compile the original code, something relevant in embedded systems, where good C++ compilers are not available to all platforms.

4. CONCLUSIONS

The combination of techniques originally not compatible with *SMP*, such as weaver based *AOP* can be achieved by separation of metaprogram compilation from the complete compilation process. Using a modified compiler to solve the metaprograms in advance, than dump the parse tree of the new program we can rebuild a program free of template and any other metaprogram structure, while keeping all the advantages of these techniques.

5. REFERENCES

[1] Andreas Gal, Wolfgan Schrder-Preikschat, and Olaf Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *Proceeding of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, U.S.A., October 2001.

[2] Gregor Kiczales. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvskyl, Finland, June 1997. Springer.

[3] Carlo Pescio. Template Metaprogramming: Make Parameterized Integers Portable with this Novel Technique. *C++ Report*, 9(7):23–26, 1997.

[4] David R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the First International Joint Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.