AOP Support for C#

M. Devi Prasad
Manipal Center for Information Science
Manipal Academy of Higher Education
Manipal – 576119
Karnataka, India
Telephone: +91- 08252-571914
devi.prasad@mahe.manipal.edu

B.D. Chaudhary
Department of Computer Science
Motilal Nehru National Institute of Technology
Allhabad – 211004
Uttar Pradesh, India
Telephone: +91-532-2541006
bdc@mnrec.ac.in

ABSTRACT

We have extended the C# compiler available under Microsoft's Shared Source Common Language Infrastructure (SSCLI) to facilitate Aspect Oriented Programming. The resulting compiler targets Microsoft .NET architecture. Our implementation introduces new ideas into the aspect language and the aspect-weaving mechanism. Our AOP extensions emulate AspectJ programming model and augment it with constructs that harness facilities provided by the Microsoft's .NET architecture. In particular, our framework allows aspect definitions to introduce 'attributes' on base C# module elements. This allows .NET runtime to provide container services transparently to marked modules and module elements.

Our aspect weaver brings novelty to the weaving phase. It allows configurable aspect ordering and selective aspect weaving. Selecting aspects of interest from a set of defined aspects and specifying suitable physical order for advice weaving is externalized in an XML file. Aspect scheduler determines a weave plan based on this specification and the weaver carries out this plan.

Modifications to the original compiler involved reorganizing the source code so that we obtain better modularization promoting reuse, or reduce coupling with the base code. In some cases, we extracted classes used as implementation helpers in the base compiler and turned them into new reusable abstractions.

In order to implement efficient traversals between abstract syntax graph and a graph representing semantics-verified method bodies, we had to extend data structures for syntax and method-body graphs. The solution introduced cyclic references across graphs. Therefore, we had to define new protocols for memory management so that these graphs representing method bodies are preserved even after their semantic checking. We altered the memory ownership scheme so that the base compiler and our extension subsystem coordinate memory management concerns.

Current (incomplete) implementation is around 2 thousand lines of C++ code over and above Microsoft's base compiler code. This implementation performs only a source to source translation. It has taken about four months for four people to make stable extensions. There is no public release of the implementation available as yet.

1. INTRODUCTION

This paper summarizes the novel features of our AOP extensions to C# language [3]. It also reports the experience gained while restructuring and enhancing a shared source compiler. Here we describe a general global view of this project, named CAMEO. The initial aim of CAMEO is to implement AspectJ like language support enabling aspect-oriented modularization in C#. Other goals include support for structural aspects that harness Common Language Runtime (CLR) features [1], incremental or partial aspect weaving, and configurable advice weaving. A preliminary source-to-source translator implementation is available for internal use. We intend to evolve this framework for exploring new ideas in AOP and metaprogramming.

Determining a collection of joinpoints in the base source involves performing a detailed control flow analysis of the code. Conducting flow analysis directly on the source text is an expensive operation in most practical cases. Since traditional compilers routinely parse source text and build Abstract Syntax Graphs (ASGs), it makes sense to make use of available infrastructure from implemented translators. In the CAMEO project, we counted on Microsoft's Shared Source Common Language Infrastructure (SSCLI) implementation to meet these requirements.

SSCLI [2] is an implementation of Microsoft's CLR architecture. Apart from the implementation of a Virtual Execution Engine and host of other tools, it includes C++ implementation of a C# language translator. The latter is a complete implementation of the ECMA standard [3] and its source code is available for modifications only for academic and research purposes. The SSCLI provided compiler generates Microsoft's Intermediate Language (MSIL) code. Because the compiler is a tiny part of the larger framework, it is inextricably tied to the infrastructure. SSCLI itself is spread around some 9000 files and the compiler source occupies nearly 200 files. We decided, very early in the project, to separate compiler code per se from rest of the SSCLI. That way, building the compiler did not require 'make'ing the entire SSCLI and it also eliminated dependencies on other tools.

The base compiler contains a collection of classes and Component Object Model (COM) components implemented in C++. It comprises lexical analysis, parsing, semantic checking, and code generation phases. The lexer reads entire source file into the memory and creates a token stream. Eventually all source files are tokenized and stored in core. Parser operates on this token stream to create an Abstract Syntax Graph (ASG). The semantic analyzer builds a Symbol Graph (SG) for namespaces, classes, methods, and other structural language-elements while performing semantic checks on types and inheritance hierarchies. It also builds an Expression Graph (EG) while binding method bodies. From there, the code generator emits MSIL executables. Totally, there are about one hundred different kinds of syntax

nodes, symbol nodes, and expression nodes. By referring to these three graphs, it is possible to regenerate the original source without any loss of information.

We turn off code generation from the base compiler and redirect control to CAMEO subsystem. CAMEO will then go through a series of phases that operate using three different data sources: the abstract graphs generated by the base compiler, aspect definition files, and an aspect configuration file. The output of CAMEO is a morph of original C# source program with aspects woven into it. Figure 1 shows the normal flow of data and control between the subsystems and phases within them.

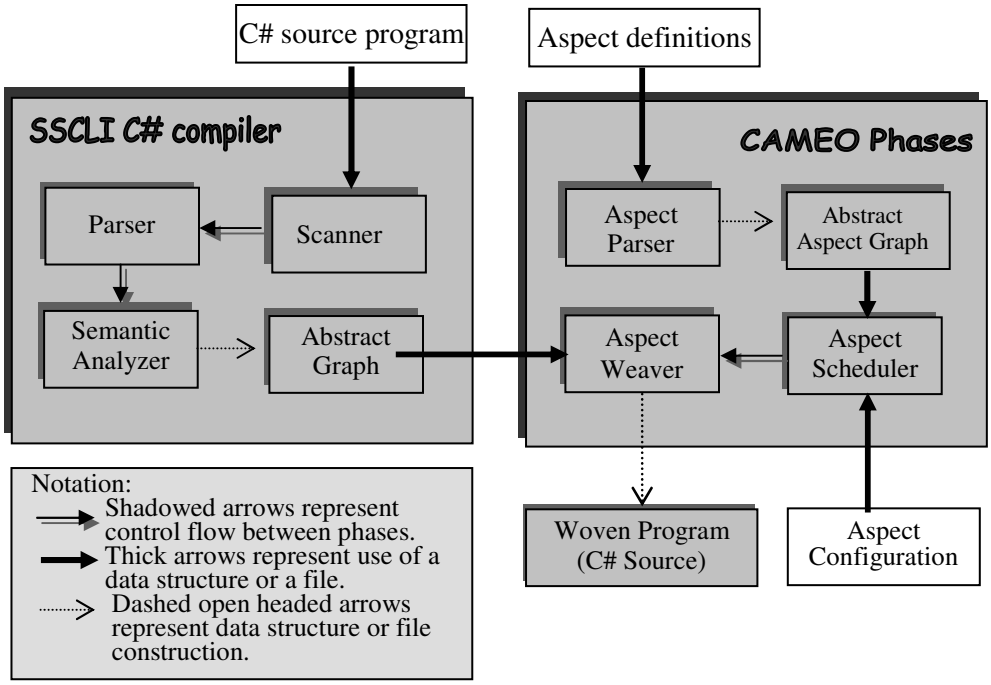

Figure 1 - Flow of control across CAMEO aspect weaver

The aspect parser builds an abstract aspect tree (AAT) from aspect definitions. This tree represents all inter-type introductions, pointcut declarations, or local member definitions found in the body of an aspect declaration. This tree is used in aspect selection and weaving stages.

CAMEO weaver takes an approach fundamentally different from that of AspectJ [4]. The latter uses '**dominates**' clause to define precedence relationships among advice encapsulated by different aspects. We feel such "concerns" about advice ordering must be separated from aspect definition per se. In CAMEO, the description of precedence relations is externalized in an XML based configuration file, much in the spirit of descriptions supplied to the popular 'make' tool. Each 'rule' in the configuration file specifies dominating rules and aspects that are part of this rule. Aspect scheduler plans a weaving order based on this information. Aspect weaver uses this plan to weave advices into the base code.

In AspectJ all aspects from an aspect definition file are automatically included for weaving. The only way to avoid particular aspects is to physically separate their definitions into different files and exclude these files from the compilation unit. In CAMEO, we have an improved method for excluding aspects from a weave step. CAMEO uses an external configuration file containing aspect composition rules. It is much simpler to specify and maintain appropriate rules for combining required aspects than to configure physical compilation units. This simplified method for separating aspect selection concern from its definition helps in building variants of a base system in a side-by-side, configurable fashion.

The rest of this paper is organized as follows: in section 2, we explain some novel features of our aspect language and the weaver. In Section 3, we discuss the challenges faced in implementing these features on top of the existing translator. In Section 4, we compare the overlapping ideas between CAMEO and a popular dynamic AOP framework. In the final section, we provide a summary of the work.

2. ASPECT LANGUAGE SUPPORT AND WEAVING MODEL

In order to maintain conformance with contemporary aspect technology, we decided to make CAMEO weaver emulate AspectJ. Moreover, C# is quite similar to Java at various levels. Hence, the syntax for aspect definition, pointcut declaration and advice declaration in CAMEO remains similar to AspectJ with minor differences. We plan to support all primitive pointcuts of AspetcJ in CAMEO. There are certain idiosyncrasies in C# and .NET that call for additional pointcuts to pick up special joinpoints and special weaver behavior. We provide a representative list of specialties in CAMEO here:

— The '**get (…)'** and '**set (…)'** primitive pointcuts in AspectJ pick up those joinpoints that access a non-private member of a Java class. On the other hand, C# has a special syntax for defining property accessors and mutators, in addition to the ordinary field accessors. In C#, property members resemble methods of a class, both in syntax and semantics. Therefore, we should treat property members of class, as well as non-private fields of a class, orthogonal. These two pointcuts, therefore, need different treatment from that of AspectJ.

— C# programs can use "unsafe" pointers. The '**unsafe**' primitive pointcut picks up joinpoints from the base code that use pointers.

— The inter-type declaration can introduce new attributes on existing classes, members of a class, return type and formal parameters of methods.

We feel these joinpoints are interesting to both designers and developers under the .NET architecture. Code using unsafe pointers could be compromising reliability for functionality. We can control tensions between safety and functionality using '**declare error**' or '**declare warning**' constructs within aspect definitions, in a case-by-case fashion.

The next three subsections bring out important ideas that distinguish CAMEO from other contemporary aspect weavers.

2.1 Structural Aspects Targeting .NET Architecture

AOP frequently uses inter-type 'introduction' mechanism to affect structural and behavioral changes in classes. An aspect definition can extend some class declaration to implement one or more interfaces, and introduce new methods and fields into that class. When components participate in specific patterns of interactions, AOP helps in implementing protocols in a non-invasive way by introducing required members and interfaces on partaking classes. Introduction in CAMEO works transparent to the target classes.

In some cases, stylistic naming conventions are strictly followed in naming methods and fields of classes. An external framework can later use reflection or static analysis techniques to provide certain services to methods that are stylistically named. Many software-testing frameworks follow this approach to automate testing. The use of declarative 'attributes' in .NET avoids some well-known problems with stylistic naming conventions [6]. Nunit Version-2 [7], a unit-testing framework for .NET languages, extensively uses attributes. Applications can also define custom attributes to annotate classes and methods and reflect upon them to provide services or modify behavior at runtime [5].

In CAMEO, we have a provision to introduce attributes on classes, methods, parameters, and fields. A statement such as the following one, defined inside an aspect declaration, introduces new attribute 'Test' on 'PushTest' method of 'Stack' class that is visible in the 'org.example' namespace.

```
introduce [Test ] on public org.example.Stack.PushTest (…);
```

Following construct introduces a new private field named 'url' into the class 'org.example.Machine'

```
introduce private String url on org.example.Machine;
```

The base compiler maintains a sequential in-core token stream representing the source program. Inter-type declarations modify physical structure of a class after the creation of token stream. Consequently, we cannot efficiently update the sequential buffer when introductions arbitrarily modify the token stream. In CAMEO, we create new abstract syntax nodes to represent the synthesized attribute or member declaration. Then we hook them into the target (which can be a class, method, formal parameter or return type) node in the original ASG. Synthesized nodes lack matching source text in the token stream.

This poses an interesting problem while unparsing, which is the last phase in CAMEO. There we need to reconstruct source text from woven ASG and EG. Because we cannot directly map synthesized nodes to text tokens, we should explicitly implement an 'Unparse()' method to generate source text from their abstract internal representation. We have generalized this technique and applied it to all types of abstract nodes. An important benefit of this approach is that in the future we can do away with the token stream making way for flexible solutions.

2.2 Configurable Aspect Ordering

CAMEO uses an XML file containing weave rules. Each rule logically stands for a concern that is implemented in terms of related aspects. In some cases, it also specifies requirements regarding the order of execution of advice. Each rule has two parts. One part lists aspects central to the concern represented by that rule. We treat a rule to 'own' aspects listed under it. The other (optionally empty) part specifies precedence relation among aspects owned by this rule, and other 'interfering' aspects. For example, a logger concern may take precedence over a security concern. A typical rule reads:

```
<rule name = 'XmlSerializer'>
  <precedence type = 'strict'>
    <dominatingRule name = 'Logger' />
    <dominatingRule name = 'SecurityManager' />
```

```
    </precedence>
    <weave>
      <aspect name = 'XmlWriter' />
    </weave>
  </rule>
```

This example says that rules 'Logger'and 'SecurityManager' dominate the rule 'XmlSerializer'. Moreover, the attribute *type = 'strict'* on the 'precedence' element specifies that the aspects listed by "Logger" (which is not shown here) must be weaved before the aspects under "SecurityManager" (again, not shown). If *type* attribute is 'lax', the weaver is free to weave the aspects under the rules 'Logger' and ''SecurityManager' in arbitrary fashion. In the example, aspect 'XmlWriter' is weaved only after the aspects owned by 'Logger' and 'SecurityManager' rules are completely considered.

The aspect scheduler computes the closure of rules starting from a 'head rule' representing the root of precedence relationships. This step should yield an acyclic directed graph. A cyclic graph implies circular dependency of rules and aspect relations, which is illegal. Consistency checks on this graph detect conflicts in aspect precedence relations. We obtain a flattened list of owned aspects from the closure of rules. This list represents a complete weave order. If there is any non-consecutive repetition of aspect names in this list, we flag the precedence requirement illegal. In the current implementation, we always assume strict ordering for aspects. A non-consecutive repetition of an aspect $\mathbf{a}_j$ across two different rules implies that at least two aspects for distinct concerns have incompatible expectation from $\mathbf{a}_j$. When $\mathbf{a}_j$ appears consecutively, it is folded into one instance.

2.3 Selective Aspect Weaving

While integrating software developed using AOP techniques, testing becomes easier if aspects are propagated in a staged manner. By guiding the weaver to deal with few selected aspects, it is possible to avoid aspect 'interference'. In CAMEO, we can introduce fresh set of rules into the configuration file, any time it is necessary to control aspect impact on the base code. These new rules should capture only the desired aspects and their inter-relationships. The command line argument to compiler should indicate the new head rule for aspect selection. As discussed in the previous section, we obtain a flattened list of owned aspects from the closure of head rule.

The aspect scheduler refers the aspect closure graph, described in section 2.2, to determine the list of aspects relevant to the current weave step. In the most general case, the scheduler constructs a list that honors aspect precedence relations. The current implementation handles only strict precedence rules, mentioned in section 2.2.

The weaver receives abstract aspect tree and precedence list from the aspect scheduler. The precedence list represents the weave order for aspects of interest. At first, the weaver considers inter-type introductions from each aspect. The weaver performs field introductions, followed by method and property introductions and then attribute introductions, in that order, on the target classes. Introductions modify the in-memory ASGs. Then the weaver processes pointcut declarations. This step performs a flow analysis of the transformed ASG in order to identify potential sites for advice weaving. Finally, the advice weaver carries out required modifications by inserting code at appropriate locations in the ASG.

3. OTHER IMPLEMENTATION CONCERNS

Some of the modifications discussed in the previous sections posed complex engineering problems. We mention these problems in this report because they represent a class of concerns that are best candidates for aspectual modularization. We hope that an understanding of such concerns helps better design of traditional software.

A careful study of the SSCLI C# translator source reveals numerous crosscutting concerns. Important among them include incremental compilation, thread synchronization (during incremental compilation), symbol table creation, memory management, compiler options influencing different phases, error reporting, executable file creation, and COM support

It is clear that the above concerns contribute to the efficiency of translator implementation. In general, efficiency concerns are scattered and tangled inside compiler's implementation. A minor modification to one feature escalates changes in many modules, either across multiple methods within a class or across class hierarchies.

For instance, the base compiler uses a class named 'CLSDREC' that maintains information necessary to declare a single class. This class has a method named 'compileMethod' that builds a parse tree for a method definition and generates the intermediate language (IL) instructions for it. For obvious efficiency reasons, its implementation releases memory held by the parse tree after IL generation. Though we have switched off the code generation phase, we would still like to exploit interior parse tree for its rich semantic content. In the base compiler, a COM class implements the interior parse tree. Each object of this type maintains references to other classes that represent source module and related abstractions. We changed the implementation so that CAMEO takes up ownership of allocated memory. This change in memory management policy manifests as two scattered blocks of code. First, we modify the flow within 'compileMethod' to avoid deallocation on successful compilation. We allow a deallocation only when compilation fails. Next, as a last step in CAMEO, we iterate every method node in the ASG and release the reference to the parse tree component it holds. This is necessary because we have to honor COM's reference counting requirements.

In some cases, we were able to refactor code from base implementation and reuse it effectively. For example, there is a class named 'CSourceText' that was used in the base compiler as an implementation helper for buffering C# source programs. This class can read

content encoded in UTF8, UNICODE or ASCII. We extracted its class declaration and implementation into different physical files and reused it for reading aspect definitions. Although this class uses an inflexible buffering strategy, its design simplicity facilitated good reuse.

4. RELATED WORK

At present, the choice of weaving strategy appears to be one of the important distinctions among different AOP languages and frameworks. Languages based on AspectJ model employ static weaving. Java Aspect Components (JAC) [8] represents an approach that exploits runtime infrastructure to provide dynamic composition of aspects. Some enhancements claimed by CAMEO overlap with that of JAC [9]. However, there are important differences in realizing these enhancements. Therefore, in this section we briefly bring out essential distinctions between JAC and CAMEO.

JAC is a powerful framework comprising of a runtime infrastructure for dynamic aspect composition. JAC does not extend the Java language. Instead it works directly on the java bytecodes. Its programming model consists of four important elements: a base object, an aspect component, an application specific weaver object, and a wrapping controller. All four are pure java objects. JAC uses Javaassist [10] to intercept requests to load a class and constructs a run-time environment suitable for composing objects with aspects. Each aspect component is written using a reflective API that helps in coordinating the execution of multiple aspects wrapped around a base object. This API also helps an aspect component to probe runtime call stack and determine method call context. The weaver object uses reflection to wrap base objects with aspect objects. An application specific wrapping controller assists in dynamically verifying the consistency among aspects, in defining precedence among aspects in a modular way. JAC derives its effectiveness from the powerful reflection based API.

In contrast, CAMEO employs static weaving, provides C# extensions, and works with source code. It does not use any runtime infrastructure or reflection, nor does it deal with bytecode weaving. The CAMEO programming model does not directly deal with consistency and precedence among aspects. Instead such issues are handled by externalized declarative specifications. These preferences have to be known at compile time.

5. CONCLUSION

CAMEO employs Microsoft's SSCLI code base for building experimental AOP infrastructure for C#. As an ongoing project, it intends to serve as a tool for advanced separation of concerns for .NET application development. The current implementation does not handle all pointcuts available in AspectJ. Nor does it support byte code level weaving. We have a few ideas on the paper for adding new joinopints to the existing repertoire of CAMEO that simplify constructing a class of structural design patterns. We have plans to refine the ideas of aspect configuration to handle lax ordering among different aspects. We also intend to replace existing in-memory token stream by a more flexible scheme so that inter-type introduction, advice interlacing becomes more efficient.

The main contribution of this project is in exploring mechanisms to separate aspect definitions from aspect weaving concerns. We have demonstrated that selective aspect weaving and configuring their weave order is both attractive and useful. We believe such separation of aspect weaving concern helps in better reuse of aspect definitions.

6. ACKNOWLEDGEMENTS

We would like to thank the CAMEO team: Aravind, Gopichand, Jagadish, and Lalitha. Thanks to our colleagues Mohan and Veena for their support at various stages. We appreciate the AspectJ mailing list members for their critical analysis of selective aspect weaving and for comparing it with the strategies employed in AspectJ and Hyper/J.

7. REFERENCES

[1] Microsoft .NET architecture and resources – www.microsoft.com/net

[2] SSCLI – http://www.microsoft.com/licensing/sharedsource/default.asp

[3] ECMA C# specification - http://www.ecma.ch/ecma1/STAND/ECMA-334.htm

[4] AspectJ download, documentation – http://www.eclipse.org/aspectj/

[5] Dharma Shukla, Simon Fell, and Chris Sells. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. MSDN magazine, March 2002 http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/AOP.asp

[6] Martin Fowler. How .NET's Custom Attributes Affect Design. IEEE Software September/October 2002. http://www.martinfowler.com/articles/netAttributes.pdf

[7] Nunit. http://www.nunit.org

[8] Java Aspect Components. http://jac.aopsys.com

[9] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In Reflection 2001, pages 1-24, 2001. LNCS 2192.

[10] Javaassist. http://www.csg.is.titech.ac.jp/~chiba/javassist