Nu: Preserving Design Modularity in Object Code

Robert Dyer Harish Narayanappa Hridesh Rajan
Dept. of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA, 50011, USA
{rdyer, harish, hridesh}@iastate.edu

ABSTRACT

For a number of reasons, such as to generate object code that is compliant with the existing virtual machines (VM), current compilers for aspect-oriented languages sacrifice design modularity when transforming source to object code by losing textual locality and intermingling concerns in the object code. Sacrificing design modularity has significant costs, especially in terms of the speed of incremental compilation. We present an intermediate language design that preserves aspect-oriented design modularity in Java byte code. We briefly describe our extensions to the Sun Hotspot VM to support the new intermediate language design.

1. INTRODUCTION

Aspect-oriented (AO) languages [6] support novel mechanisms for separation of traditionally non-modular concerns. The problem that we address in this work is that AO compilers sacrifice this separation of concerns, while transforming source code to object code. The design modularity that AO languages bring is lost in object code (see Figure 1). There are real opportunity costs to losing design modularity in object code. One particular cost comes from the resulting complicated mapping between modularized concerns at the source code level and the object code fragments. Compilers currently have to perform this forward mapping. Debuggers and analysis tools do this mapping in reverse.

These costs are visible in the performance of incremental AO compilers. The best AO compilers available today take significantly more time and memory compared to their object-oriented counterparts for incremental compilation. Techniques such as Smartest Recompilation [11] have shown improvement to incremental compilation times; however, these techniques are not always applicable to the AO paradigm. Recently, Lesiecki [7] observed that incremental compilation using the AspectJ compiler for 700 classes and 70 aspects usually takes at least 2-3 seconds longer than near instant compilation using a standard Java compiler.

These are not new problems. The compilers for the past two successful modularization techniques, structured programming [2] and object-oriented (OO) programming [1], have shown similar traits. In the absence of a call instruction in the instruction set architecture, early implementations of a procedural language compiler would translate modularized procedures into a monolithic set of instructions by in-lining the procedure bodies.

These programs would get the benefits of separation of concerns in the analysis, design and implementation phases, however, in later phases, such as incremental compilation and debugging, the benefits were lost with the loss of design modularity. For example, consider changing an in-lined procedure. This change will have to be reflected at all call sites in the object code because the method is now in-lined by the compiler. This in turn makes incremental compilation of procedures complicated and inefficient. Similarly, the reader is encouraged to think about incremental compilation of OO programs into intermediate code languages that do not support *invoke virtual* as an instruction.

Our key observation is that for structured programming and OO programming, the invention and refinement of intermediate languages (or instruction set architectures) helped preserve design modularity in object code and brought the benefits of separation of concerns to post-compilation processes such as incremental compilation, debugging, etc. Based on this observation, we present an approach for preserving design modularity in object code for aspect-oriented programs. Our approach consists of an improved intermediate language model and a virtual machine design to support the new model. We show in our presentation that representation of concerns remains modular in our intermediate language model.

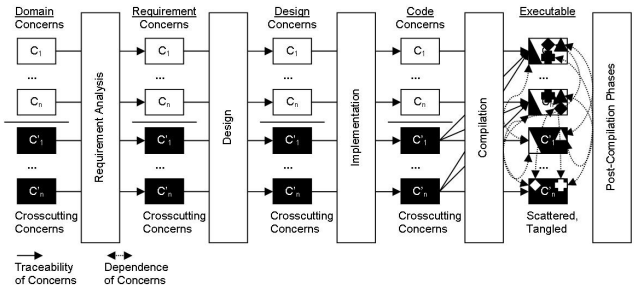

Figure 1: Tracing Concerns through the Life Cycle

FSE '06 Portland, OR

2. OUR APPROACH

To preserve the separation of concerns after AO compilation, we proposed a new intermediate language model called *Nu* [9, 3, 8]. The *Nu* language model adds two new primitives: *bind* and *remove*. These primitives expect a *pattern* and a *delegate* as arguments. Similar to *pointcuts*, the *pattern* serves to select a subset of the *join points* in the program. *Pointcuts* and *join points* have the same meaning as in AspectJ-like languages. The *delegate* or the *delegate chain* – similar to an AspectJ advice [5], specifies a list of methods that is to execute at these *join points*. The *bind* primitive associates the supplied *delegate* with the *join points* matched by the corresponding *pattern*. A successful *pattern* match with a *join point* results in the related *delegate chain* being invoked when the program execution reaches that *join point*. The *remove* primitive eliminates this association. Our primitives are similar to the previous work of Rajan and Sullivan [10]. They showed that extending OO classes with a new declarative construct, *binding*, unifies OO and AO program design, enabling the modularization of hierarchical integration concerns. Our experiments described elsewhere [3] have shown that the *bind* and *remove* primitives are able to support most constructs in existing aspect languages.

Without support for *bind*, an AO compiler inserts calls in other modules to produce the desired behavior when generating the object code, sacrificing the semantic and syntactic separation of the concerns defined by the aspects. With support for *bind*, the compiler translates the pointcuts defined by the aspect into instructions to construct equivalent *patterns*. It then generates instructions to bind the delegate to the join points defined by the pointcuts. The translation with support for *bind* preserves the textual locality of the aspects in the object code. The base classes are free of scattering and tangling and any changes to their corresponding source code will be local to their own byte code. Similarly, any changes to the source code of aspect classes will be local to their corresponding byte code. This maintains the separation of concerns for *Nu* programs at the byte code level.

As a proof of our concept, we realized the *bind* and *remove* primitives by providing support for these at the Java Virtual Machine (JVM) level and a corresponding Java API. These primitives accept a *pattern* and a *delegate* as their arguments. After compiling both the base classes and the aspects (aspects here are first class entities, containing the bindings), there is no scattering or tangling in the generated class files. Instead of weaving instructions in at various *join points* in the Java byte code of other classes, the instructions are localized to the aspect's byte code.

At runtime, the *join points* in the program are matched against previously bound patterns. To accomplish this, the Java Hotspot VM was modified by adding an implementation of a join point dispatcher and changing the JVM code to notify the dispatcher of join point executions. To reduce the overhead incurred when pattern matching, the JVM attempts to limit the number of calls to the join point dispatcher based on analysis performed at class load time. This analysis efficiently determines if a join point possibly matches a bound pattern. The dispatcher then attempts to match the join point against all bound patterns in the join point's delegate chain. For each matched *join point*, the corresponding *delegate chain* is invoked. Our current dispatching mechanism is inefficient; however, we are exploring methods to decrease the overhead.

3. CONCLUSION AND FUTURE WORK

The new invocation mechanism attempts to improve conceptual integrity of AO programming models. We are currently investigating other related areas to improve the existing infrastructure - better run-time support for the existing execution model, support for high-level AO constructs in the intermediate language [4], providing a formal semantics for *Nu*, etc. The decoupling between language compilers and the virtual machine achieved by the interface provided by our invocation mechanism also has the potential to enable independent research in these areas. Simpler aspect language designs and compiler implementations might be realized without spending significant time on the optimization of the underlying AO execution models. Novel optimization mechanisms for the underlying execution models can be developed independent of the language design, as long as it conforms to the interface.

4. REFERENCES

[1] O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[2] E. W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[3] R. Dyer, H. Narayanappa, Y. Hanna, and H. Rajan. Nu: Improving aspect oriented incremental compilation. In *Submission*.

[4] R. Dyer and H. Rajan. Modular compilation strategies for aspect-oriented constructs. In *Submission*.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: 15th European Conference on Object-Oriented Programming*, pages 327–353, Budapest, Hungary, June 2001.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[7] N. Lesiecki. Applying AspectJ to J2EE application development. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, 2005. ACM Press.

[8] Nu web site. http://www.cs.iastate.edu/∼nu.

[9] H. Rajan, R. Dyer, Y. Hanna, and H. Narayanappa. Preserving separation of concerns through compilation. In L. Bergmans, J. Brichau, and E. Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies (SPLAT 06), A workshop affiliated with AOSD 2006*, March 2006.

[10] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[11] Z. Shao and A. W. Appel. Smartest recompilation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 439–450, New York, NY, USA, 1993. ACM Press.