Nu: Enabling Modularity in Multilingual, Multienvironment, Distributed Systems

Hridesh Rajan
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50010, USA
hridesh@cs.iastate.edu

ABSTRACT

The contribution of this work is a novel aspect-oriented programming model that we call Nu. The Nu programming model adds only one new concept, join point dispatch, to the object-oriented programming model. No new programming language constructs are added. The constructs in existing aspect languages are expressed in terms of join point dispatch resulting in a significant simplification of aspect languages. We make two claims about the potential benefits of our approach. First, that it will enable transparent modularization of even those crosscutting concerns that transcend the language and environment boundaries. Second, that it will simplify the AOP language model resulting in the ease-of-use, ease-of-learning, and reduced cost to build supporting tools.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features —classes and objects; Modules, Packages

General Terms

Design, Languages

Keywords

Bindings, Classpect, Unified Aspect Language Model

1. MOTIVATION

In aspect-oriented programming terminology, a crosscutting concern is a dimension in which a design decision is made and whose realization in traditional designs leads to code fragmented across a system and intermingled with code for other concerns. Security policy enforcement, cross-module optimization, execution tracing policy, etc, are some examples of crosscutting concerns. Most software systems today are implemented as a collection of components that interoperate with each other, but that may execute in separate address space, may execute on separate software/hardware platforms, and may be written in separate high-level languages. In such systems, the realization of a crosscutting concern leads to code that is fragmented across address spaces, software/hardware platforms, and high-level languages.

Aspect-oriented programming (AOP) [6, 11] has shown the potential to improve the ability of software architects to devise more effective modularizations for some crosscutting concerns [15, 3]. Current AOP mechanisms can modularize crosscutting concerns that transcend language and system boundaries by using a combination of languages and libraries; however, as Benton et al. argue, such solutions fail to bring benefits such as better compiler optimization, code generation, bug analysis, etc. [2]. Ironically, these embedded, pervasive, distributed, system types—coupled with the internet and the web—are in fact driving the invention, refinement, and adoption of AOP.

Not too long ago, in response to a similar challenge to make it easier to write accessible components and applications in any language and to use them across languages, the Common Language Infrastructure (CLI) was proposed by Microsoft, HP, IBM and others [5]. What is needed now is a similar interoperable approach to AOP in which aspects written in multiple high-level languages can be executed in different system environments without the need to rewrite these aspects to account for the unique characteristics of those languages and environments.

We are not the first to recognize the need for language interoperable aspects. Among others, Lafferty and Cahill [12] proposed a language independent notion of aspects that exploits the multi-language support of Common Language Infrastructure (CLI). Their approach adopts the AspectJ language model [10] to CLI. Their approach, however, relegates the responsibility of aspect-component composition to second-class XML scripts. Our approach, like AspectJ, aims to express these compositions as part of the language. In addition, they do not provide support for instance-level advising, first-class aspect instances, and environment interoperability.

The rest of this paper is organized as follows. The next section describes a motivating example for our work. Section 3 describes the proposed language model. Section 4 describes the potential impacts of this research and identi-

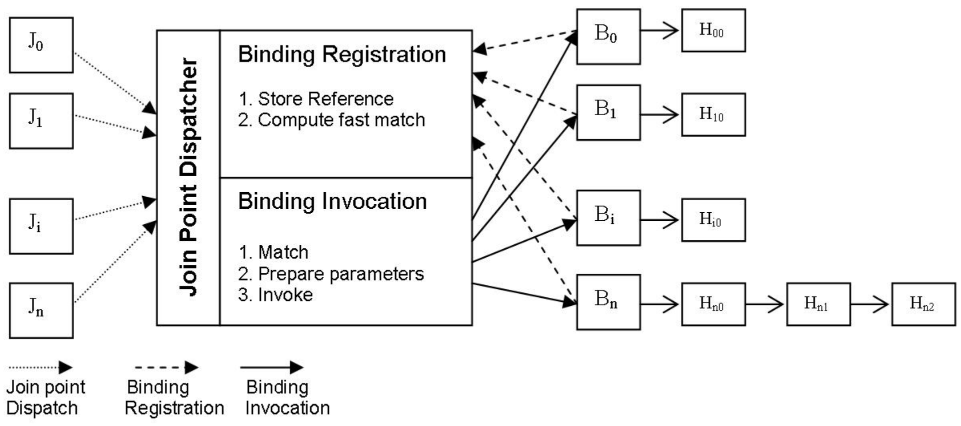

Figure 1: The Overview of Nu

fies future investigations.

2. MOTIVATING EXAMPLE

To make the problem concrete, let us consider an example system. Our example system is a smart-home system, designed for assisted living, and implemented as a collection of components executing on sensors, handheld devices, and web services. The components executing on sensors collect data about the environment. They are programmed in C and execute in the TinyOS environment [9]. The components executing on handheld devices are used by the clients to interact with the smart-home system, e.g. to change the average temperature of the home. These components are programmed in Java and execute in the JAVA Virtual Machine (JVM). The web services are programmed in C#/ASP.NET and run in the .NET Framework.

Let us consider the implementation of a simple crosscutting concern, execution tracing policy, for this system. In a traditional design, the realization of execution tracing policy concern is fragmented across and intermingled with the code for components written in three different languages (C, Java, and C#) executing on three different software/hardware platforms (TinyOS, JVM, and .NET Framework). The implementation of this concern also varies across these components. In C#, the *System.Console* library is used to write the execution trace, in Java, the *System.out* library is used, and in C on TinyOS, the trace is output on a port.

Current aspect-oriented approaches can modularize this concern using a combination of languages and middleware. For example, one may use AspectJ [10], Eos [13], and AspectC [1] to modularize the execution tracing policy in Java, C# and C partitions of the system respectively. The execution trace generated by these three modularizations can be accumulated manually or by adding an infrastructure to gather execution trace across the three environments. This solution works, however, it is less then adequate. First, we haven't yet achieved a complete modularization of the crosscutting concern. Instead, the concern is still spread across n components of the system, where n is the number of different aspect-oriented extensions employed. Second, these n components implement the same functionality, essentially requiring duplication of intellectual efforts. Third, using a combination of languages and libraries makes it unnecessarily hard to apply advanced analysis techniques to the concern as a whole. Instead, only parts can be locally optimized.

3. NU PROGRAMMING MODEL

To address the interoperability problem, in this work we propose a new approach to aspect-oriented programming that we call *Nu*. We hypothesize that to achieve language interoperability and to build a language interoperable infrastructure it is necessary and sufficient to provide a precisely specified invocation mechanism that allows representation of crosscutting concerns as modular units of program design. The aspect-oriented constructs in the high-level languages can be expressed in terms of these primitives without compromising their expressiveness.

Our previous work on unified aspect model emulated by Eos [7], an aspect-oriented extension of C# for Microsoft .NET Framework, motivates this hypothesis. We showed that *aspect* as a syntactic category can be eliminated in favor of a more expressive notion of class called *classpect* that has an additional construct called binding [14]. This single construct allows modularization of crosscutting concerns. We informally defined binding as a mechanism to select a subset of join points in the execution of the program and associate a method to execute at those points. A join point in AOP terminology is a point in the execution of the program. The subset of join points selected by the binding are called subjects of the join point. The method that is associated by the binding to execute at these join points is called the handler of the binding.

In Nu, we take this idea to the next level. Nu adds only one new concept to the underlying language semantics (also called base language). In Nu, all join points invoke a *Join Point Dispatcher (JPD)* as shown in Figure 1. A *JPD* is similar to event dispatcher used in the modeling of implicit invocation systems [4]. There is only one instance of the JPD in one address space. As shown in Figure 1, all join points in an address space invoke the JPD. We call this phenomenon *join point dispatch*, similar to *event dispatch* in implicit invocation systems.

The Eos style binding are represented as object-oriented classes that inherit from *Nu.Runtime.IBinding* interface. A binding's interface provides a function to match the subject join points, a function to invoke the handlers, and functions to bind and remove object instances from the binding's association. A binding may optionally also implement the *FastMatch* interface (not shown in Figure). The *FastMatch* provide a function that returns a unique hash for the binding to optimize dynamic join point matching. As shown in Figure 2, the before, after, and around bindings in Eos are implemented as three object-oriented classes that implement the *IBinding* interface.

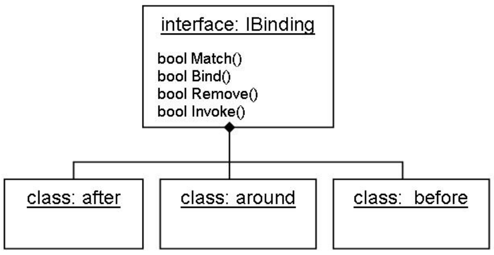

Figure 2: The Binding Hierarchy

As shown in Figure 1, all bindings in an address space register with the JPD. On registration, the dispatcher stores a reference to the binding and computes the *FastMatch* of the binding, if the binding provides that interface. On subsequent join point dispatch, a match with all the registered bindings is performed. If the join point matches a given binding, the join point dispatchers calls the invoke method of that binding. The binding may in turn invoke it's handlers.

As of this writing, the Nu compiler is implemented as post processor for .NET Framework. The post processor takes a .NET assembly [1] and instruments all language-defined join points in the assembly to construct an object of type *Nu.Runtime.Joinpoint*. This object represents the reflective information about the join point. A call to the *Join Point Dispatcher* is also inserted at all join points. Instrumenting all join points is inefficient compared to the techniques used by AspectJ and Eos compilers. These compilers only instrument the join points that are potentially of interests to bindings. In future, we will implement optimizations to reduce this overhead.

Our approach is independent of base languages. This language independence comes from expressing aspect-oriented constructs in terms of join point dispatch. The bindings in Eos are now represented as object-oriented classes that implement interfaces to match join points and invoke handlers. The advice in AspectJ-like languages can be implemented as a combination of a binding and a method, as in Eos. Any combination of .NET languages can be used to write aspect-oriented programs. For example, in a polyglot system that uses VB.NET and C#, a programmer can write bindings in either of these languages or an entirely different .NET language. This language independence is similar to what achieved by Weave.NET [12], however, adopting an improved language model for aspect-oriented programming brings the additional benefits of environment independence.

4. IMPACTS AND FUTURE WORK

We speculate that the proposed research program will have significant impacts on both the theory and the practice of aspect-oriented programming. By abstracting the enabling mechanism behind the language run-time, we will be able improve the modularization of crosscutting concerns across the language and environment boundaries. The advantages would be similar to those of the remote procedure call, where the caller is syntactically unaware of the location of the executing method. By analogy, transparent modularization of crosscutting concerns, which transcend the language and environment boundaries, will become possible.

The aspect language models today include several non-orthogonal, asymmetric and irregular constructs. By eliminating all aspect-oriented constructs, for example, aspect, pointcut, advice, etc, in favor of a single richer invocation mechanism and extensible quantification mechanism, we hope to bring significant simplification and conceptual unity to the programming model without losing expressiveness. This simplification will also bring ease-of-use and ease-of-learning to aspect-oriented methods. An aspect-oriented program will look just like an OO program with the exception of the use of the new implicit invocation mechanism. It will also make it much easier to build tool support for the AO languages. By decreasing the size of the core language and expressing new extensions in terms of a language for which infrastructure exists, this research will make it possible to quickly realize production quality tools such that the benefits of the aspect language designs can be observed in real software development projects.

We foresee the following directions for future investigations:

1. *Infrastructure Extension*: In future, we will implement a post-processor for Java similar to .NET Framework. Introducing the *Join Point Dispatcher* as the mediating component between join points and bindings enables us to modularize crosscutting concerns across environments. The key challenge in this direction is to define an interface between two standalone instances of JPD. For example, to modularize crosscutting concerns across JVM and .NET Framework, an instance of JPD will be deployed for each environment. This instance deployment will be transparent. These JPD instances will interface with each other to dispatch join points in JVM components to .NET Framework components and vice-versa. To extend *Nu* to a new environment, a post processor for that environment and an implementation of the JPD for that environment is needed. A technique similar to Gray and Roychoudhury [8]'s approach, to generate aspect-oriented compilers for various programming languages, could be used.

2. *Binding Semantics*: The second objective of this research is to develop a semantics of bindings as well as a language independent notion of binding in terms of the language model developed in the first stage of this research. The key challenges in this dimension is to develop an extensible as well as interoperable notion of aspect-oriented quantification, and to account for compile-time, run-time, instance-level [13], and type-level associations of bindings with objects.

3. *Robust Infrastructure*: The third objective of this research is to develop a robust infrastructure to support

[1] An assembly is a .NET equivalent of an executable.

these ideas. We will collaborate with domain experts from various fields such as distributed, embedded, and real-time systems, sensor networks, etc. to understand the unique characteristics of these environments and to develop corresponding realizations of the *Join Point Dispatchers* for these environments.

4. *Integration with Runtime Environments*: We will also including binding primitives in the intermediate language itself, including the dispatch mechanism in the language runtime, and providing high-level language compilers that translate aspect-oriented constructs into the modified intermediate language.

5. REFERENCES

[1] AspectC web page. http://www.cs.ubc.ca/ labs/spl/projects/aspectc.html.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[3] A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, New York, NY, USA, 2004. ACM Press.

[4] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. *SIGSOFT Software Engineering Notes*, 23(6):209–21, Nov. 1998.

[5] ECMA. *Standard-335: Common Language Infrastructure (CLI) Specification, Third Edition*, June 2005.

[6] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[7] Eos web site. http://www.cs.virginia.edu/ eos.

[8] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[11] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.

[12] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[13] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[14] 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.

[15] D. Sabbah. Aspects: from promise to reality. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 1–2, New York, NY, USA, 2004. ACM Press.