Modularizing Crosscutting Concerns with Ptolemy

Hridesh Rajan[1] Sean Mooney[1] Gary T. Leavens[2] Robert Dyer[1] Rex D. Fernando[1]
Mohammad Ali Darvish Darab[1] Bryan Welter[1]

[1]Department of Computer Science
Iowa State University
{hridesh,smooney,rdyer,fernanre,ali2,bawalter}@iastate.edu

[2]Department of EE and Computer Science
University of Central Florida
leavens@eecs.ucf.edu

Abstract

In this demonstration we show our language Ptolemy, which allows for separation of crosscutting concerns while maintaining modular reasoning. We demonstrate the benefits of Ptolemy over existing aspect-oriented languages and implicit invocation designs. Ptolemy's quantified, typed events provide a flexible quantification mechanism that acts as a declarative interface between object-oriented code and crosscutting code. Events are announced explicitly and declaratively.

Event types allow for compile-time errors and avoid the fragile pointcut problem of aspect-oriented languages. The interface provided by event types also allows for modular reasoning, without considering all aspects in the system. The declarative event announcement allows avoiding writing tedious and error-prone boiler-plate code that implicit invocation designs require.

We demonstrate several realistic examples that showcase the features of the Ptolemy language and show use of Ptolemy's compiler. The demonstrated compiler is built on top of the OpenJDK Java compiler (javac), providing full backwards compatibility with existing Java sources as well as ease of integration into the existing tool chains. We show how to integrate the compiler into both existing Ant and Eclipse builds.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules and Packages

General Terms Design, Languages

Keywords modular reasoning, aspect-oriented programming languages, implicit invocation, translucid contracts, aspect-oriented interfaces, Ptolemy

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

1. Background

Maintenance and evolution of software systems is of extreme importance in software engineering. In order to evolve and maintain systems easily, a clear traceability of concerns from requirements to code must exist. Certain concerns lack such traceability due to their scattering across several modules and tangling with other concerns. Such concerns are called *cross-cutting concerns* in aspect-oriented (AO) terminology [4]. Being able to modularize such concerns is an important problem investigated by AO techniques as well as implicit invocation (II) techniques [6].

The key idea behind both AO and II techniques is to decouple components and allow for their composition at runtime using *events*. For example, in the observer design pattern, which is an adaptation of II techniques [6], the participants (*subjects* and *observers*) are decoupled in both the design and code and then composed at runtime. The subjects dynamically announce events which observers dynamically register event *handlers* (methods) that are invoked implicitly after the events are announced. This ensures that subjects are independent of any observers and allows for separate maintenance and evolution.

In AO languages such as AspectJ [5], the events are predefined by the language. Certain standard events, such as method calls, in the program's execution are provided. In AO languages, these events are announced implicitly and handlers are declaratively registered to sets of these events. This process is called quantification [3].

II techniques have two distinct advantages over AO techniques. First, all events are explicitly announced, which aids reasoning about modules announcing events as all points where such announcement may occur are explicitly marked. Second, event announcement is generally more flexible as any arbitrary point may announce an event.

AO techniques have advantages over II techniques as well. First, compared to II techniques the implicit event announcement in AO helps automate and decouple event announcers and handlers. Second, since modules announcing events do not explicitly name handlers, the handler code remains syntactically independent of all announcement code.

2. Ptolemy

The Ptolemy language [7] takes advantages from both implicit-invocation (II) and aspect-oriented (AO) techniques. It has three main design goals:

- Enable modularization of crosscutting concerns while maintaining the encapsulation of object-oriented code,
- enable a well defined interface between the crosscutting and object-oriented concerns and
- enable separate type-checking, compilation and modular reasoning of crosscutting and object-oriented code.

Achieving these goals using AO languages in the style of AspectJ is difficult. First, knowing for certain if advice may apply at a point in the code is difficult, as such potential locations occur frequently. At every such possible point, programmers must reason and account for the effects of all applicable advice. Second, in order to reason about control flow, programmers would have to reason about all potential control effects of advice at that location, including how different advice might interact with each other.

3. Benefits of Ptolemy

The event types provided by Ptolemy's design provides a declarative interface between the object-oriented and crosscutting code. Event types define the type for all announced events as well as the context information available during event announcement. Announcing events is declarative and explicit. The language design provides several software engineering benefits.

- Explicit event announcement aids in modular reasoning.
- The declarative event announcement syntax saves programmers from writing tedious and error prone boilerplate code. It also allows the compiler to statically check and optimize the event announcement code.
- Event types are statically checked during compilation, which avoids the fragile pointcut problem associated with AO techniques.
- Event types allow for specifying a contract between subjects and observers. Such translucid contracts [1, 2] expose some details of the observers, which allows programmers to understand an upper-bound on the behavior of subjects and observers by only inspecting the event type.
- Quantification over subjects does not require enumerating the subjects, which decouples the observers from subjects.

Ptolemy allows observers to declaratively express interest in certain events in the system. The bindings allow observers to refer to subjects without becoming name dependent on the subjects. This name independence allows for separate and independent maintenance and evolution of the subjects and observers.

4. Demonstration Overview

This demonstration showcases the features and benefits of the Ptolemy language through several realistic examples. The current infrastructure for developing Ptolemy programs is also demonstrated, including syntax highlighting in Vim/Emacs and use of Ptolemy's compiler. The compiler is built on top of the standard OpenJDK Java compiler (javac) and is fully backwards compatible with pure Java programs. Use of the compiler with both Ant and Eclipse is demonstrated.

5. Presenter Biographies

Hridesh Rajan is one of the two original authors and creators of the Ptolemy language. He has extensive experience in separations of concerns techniques and co-developed the aspect-oriented language Eos. He has given previous demonstrations on the Ptolemy language at AOSD'10, FSE'10 and ECOOP'11. He also taught a half day tutorial on the Ptolemy language at AOSD'11.

Robert Dyer helped develop the original research compiler for Ptolemy. He also was the lead researcher on an empirical evaluation of the Ptolemy language which involved developing Ptolemy versions of two medium sized, open source programs (MobileMedia and Health Watcher).

Acknowledgments

This work is supported in part by the US National Science Foundation (NSF) under grant CCF-10-17334 to Hridesh Rajan and grant CCF-10-17262 to Gary T. Leavens.

References

[1] M. Bagherzadeh, H. Rajan, and G. T. Leavens. Translucid contracts for aspect-oriented interfaces. In *FOAL '10*.

[2] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *AOSD '11: 10th International Conference on Aspect-Oriented Software Development*, March 2011.

[3] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA 2000*.

[4] G. Kiczales *et al.* Aspect-oriented programming. In *ECOOP '97*.

[5] G. Kiczales *et al.* An overview of AspectJ. In *ECOOP '01*.

[6] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91*.

[7] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, July 2008.