Abstract

In previous work, we and others have studied how aspects can implement important cross-cutting concerns, such as runtime monitors, security monitors, and other security primitives. It is hard to design aspects that implement such concerns correctly. Therefore, once written, one desires to reuse the according aspect definitions for other systems.

In current aspect-oriented systems, however, aspects usually carry, through their pointcuts, explicit references to the base code. Those references are fragile and give up important software engineering properties such as modular reasoning and independent evolution of aspects and base code, hence hindering aspect reuse. A well-studied solution to this problem is to separate base code and aspects using an intermediate interface abstraction.

In this keynote talk, I will show that previous approaches to solving the problem for AspectJ fail at restoring modular reasoning because they do not provide modular type checking; programs can fail to compose when woven, even though their interfaces are compatible. As I will show, the approaches fail for different reasons. Some represent join points as structs or objects, which breaks lexical scoping. Others lack important information in join point type descriptors, which precludes Java-like typing guarantees.

I will report on a novel abstraction called Join Point Interfaces (JPIs), which, by design, supports modular reasoning and independent evolution by providing a modular type-checking algorithm [1, 2]. JPIs further offer polymorphic dispatch on join points, with an advice-dispatch semantics akin to multi-methods. As I will show, our semantics solves important problems present in previous approaches to advice dispatch.

We have fully implemented JPIs as an open-source extension to the AspectBench Compiler. A study on existing aspect-oriented programs of varying sizes and domains supports our major design choices and reveals potential for exploiting polymorphism through non-trivial join-point type hierarchies. However, as the study also reveals, our current language design is not yet perfect, and thus further work is needed.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs

General Terms Design, Languages

Keywords Aspect-oriented programming, modularity

References