Aspects Made Explicit for Safe Transactional Semantics

Kevin Hoffman
Purdue University
kjhoffma@cs.purdue.edu

Patrick Eugster
Purdue University
peugster@cs.purdue.edu

1. Introduction

Transactions are the most widely employed concurrency control and reliability mechanism, enjoying many different incarnations. Language support typically provides elegance and safety but may miss interoperability and flexibility achieved with libraries. In the expectation of balancing these benefits, many authors have recently elaborated on the use of aspect-oriented programming (AOP) [5] to implement transactions. AOP has been proposed as a generic method for dealing with issues such as concurrency by capturing these with respective aspects separately from the main application logic. However, AOP pushes separation to the point of achieving obliviousness in the main application, which has been shown to introduce safety issues [7], allowing, for instance, transactional semantics to be violated in ways invisible to the programmer. We propose a pragmatic change in the way aspects are joined with the base application code and describe corresponding modifications to the AspectJ language. These changes increase safety by facilitating semantic coupling between the base code and aspects where necessary, and apply also to AOP solutions for other reliability-related issues.

2. Background

Transactions are an important building block for reliable systems, providing semantics for correctness in the face of concurrency and failures. Mechanisms for implementing transactions include specific language constructs, macros, and libraries. The Argus [8] language, for instance, defines constructs to implement location transparency, durability, serializability, and atomicity. Because the programmer explicitly uses these constructs, the language can uniformly and nearly automatically apply "transactionalizing" mechanisms according to specific rules that maintain transactional semantics, even with more complex features such as nested transactions and checkpointing. Such an approach provides safety and elegance, yet lacks interoperability, as integrating with components written in other languages requires high-level communication mechanisms. This approach also lacks flexibility, requiring new language releases or many optional constructs, which tends to bloat syntax.

Aspect-oriented programming (AOP) [5] has been proposed as an alternate means to implement transactions. AOP is a technique that minimizes redundancy by separating cross-cutting concerns – code with a common purpose scattered throughout the program – into logical units called *aspects*. AspectJ [4] is an AOP language that extends Java. Many authors have proposed AOP and in particular AspectJ to implement transactions in order to combine the safety and elegance of language approaches with some degree of flexibility and interoperability.

In AspectJ, cross-cutting concerns (described by *advice*) are *woven* into the base code at implicit structural points (called *join points*). Advice describe where exactly to inject their logic via parameterized collections of join points (*pointcuts*). AspectJ compiles to standard Java bytecode able to run on any compliant Java virtual machine.

3. Explicit Join Points

The downside to existing AOP solutions is that they do not enforce safety; in fact, it is a declared goal of AOP to achieve *obliviousness* (i.e., the base code is not aware that it is being advised). A detailed illustration, through AspectJ, of safety issues arising in the case of transactions can be found in [7].

We propose a radical change in the way aspects are joined with the base code by adding a new type of join point – *explicit join points* (EJPs). EJPs have a signature definition as part of the (possibly **`abstract`**) aspect definition. This signature defines the properties, parameters, and constraints relating to the join point. The base code uses EJPs much like method calls. EJPs may also be scoped, allowing the base code to define new structure that can be assigned specific semantics (e.g. the scope of a transaction). Through EJPs the base code informs aspects how it should be advised.

Figure 1 illustrates how EJPs could be used to implement transactions. This example focuses on the increased safety gained via EJPs.

```
aspect Transactionalizer {
  scoped joinpoint jpTrans(Serializable[] objects)
      throws CommitException;
  pointcut pcTrans(Serializable[] objects):
    joinpoint(jpTrans) && args(objects);
  before(Serializable[] objs): pcTrans(objs) {
    TransMan.begin(objs, ...);
  }
  after() returning: pcTrans() {
    TransMan.commit(...);
  }
  after() throwing: pcTrans() {
    TransMan.abort(...);
  }
}

class Agent implements Serializable {
  void createTrip(Person p, Flight f, Hotel h) {
    boolean bCharged = false;
    try {
      Serializable[] objects = {this, p, f, h};
      Transactionalizer.jpTrans(objects) {
        f.ReserveSeat(p);
        f.ReserveRoom(p);
        m_CC.Debit(p.getCC(), ...);
        bCharged = true;
      }
    } catch (CommitException e) {
      if (bCharged)
        m_CC.Credit(p.getCC(), ...);
    }
  }
}
```

Figure 1. Psuedo-code for Transactions with EJPs

4. Assessment

In the example, the `jpTrans` join point uses the **`throws`** clause, requiring base code to handle or throw the `CommitException`, which was not previously possible. This increases safety in the example because explicit compensation is required for the credit card transaction. Because `jpTrans` is scoped, the base code can explictly define transaction scope, increasing safety [7]. `jpTrans` has one parameter, requiring the base code to communicate explicit information to the aspect(s) advising the join point. Parameters could also be used to pass information from advice to base code or between aspects advising that join point.

The above example helps to illustrate three fundamental differences between EJPs and method calls: First, method calls exhibit a one-to-one relationship between caller and callee; EJPs allow several aspects to apply advice at that point. For example, one could implement transactions by separating atomicity, isolation, and durability aspects along the lines of [2]. Second, the full expressive power of pointcuts apply to EJPs, bringing us all the benefits of AOP/AspectJ (namely, minimized redundancy related to cross-cutting concerns and the ability to weave at run-time). Third, EJPs may be scoped, further structuring code in ways that can improve elegance and safety; local variables do have scope but cannot execute logic when the scope ends nor attach constraints related to the scope.

5. Related Work

Many authors have suggested improvements to AOP to improve safety. Gudmandson and Kiczales [3] for instance suggest *pointcut interfaces*, providing a new logical layer between the aspect and the base code, decoupling the two.

Aldrich builds on this work by proposing *open modules* [1], defining the concept of a “package” that explicitly exports functions and pointcuts, and they prove that package semantics are implementation independent.

Kiczales and Mezini [6] propose *aspect-aware* interfaces that allow for unrestricted use of AOP in a manner that is compatible with modular reasoning. Our approach provides safety without global analysis via an enhanced compiler.

6. Conclusions and Future Work

We introduced explicit join points, which leverage the full expressive power of AOP, yet also provide enhanced safety and functionality in places where there is a semantic coupling between cross-cutting concerns and the base code. We plan to explore additional explicit join point constraints that will increase program safety. Complex examples will be developed, and the benefits and drawbacks of our approach will be analyzed in detail. We are enhancing an AspectJ compiler to support our proposed extensions.

References

[1] J. Aldrich. Open Modules: Modular Reasoning About Advice. In *ECOOP'05*, pages 144–168, 2005.

[2] A. Black, V. Cremet, R. Guerraoui, and M. Odersky. An Equational Theory for Transactions. In *FSTTC'03*, pages 38–49, 2003.

[3] S. Gudmundson and G. Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *ECOOP'01 Workshop on Advanced Separation of Concerns*, 2001.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP'01*, pages 327–353, 2001.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, pages 220–242. 1997.

[6] G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *ICSE'05*, pages 49–58, 2005.

[7] J. Kienzle and R. Guerraoui. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *ECOOP'02*, pages 37–61, 2002.

[8] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.