Checking API Protocol Compliance in Java

Kevin Bierhoff

Carnegie Mellon University

http://www.cs.cmu.edu/∼kbierhof/

Abstract

Reusable APIs often define usage protocols. The author previously developed a sound and modular type system that checks compliance to typestate-based protocols while affording a great deal of aliasing flexibility. This paper focuses on making these ideas available in tools for mainstream object-oriented languages and evaluating their practical effectiveness.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Languages, Verification.

1. Introduction

The use of libraries and in particular the enormous "standard" libraries included with most programming languages is abundant. Such APIs often define *usage protocols* that API clients must follow in order for code implementing the API to work correctly. Usage protocols define legal sequences of method calls on objects. For example, one can retrieve rows of a database query result and then close it, but one cannot retrieve more rows after the result was closed.

The goal of my work is to give developers comprehensive help in following *and implementing* API protocols expressible using *typestates* (Strom and Yemini, 1986). I previously developed a sound (no errors missed) and modular (every method is checked separately), but highly non-deterministic, type system based on typestates for a core object-oriented calculus that offers a great deal of flexibility in reasoning about aliased objects (Bierhoff and Aldrich, 2007). Unlike existing modular protocol checkers (e.g. DeLine and Fähndrich, 2004) we do *not* require precise tracking of all aliases to an object, and in contrast to modern program verification tools (e.g. Barnett et al., 2004) we do not impose an ownership discipline on the heap.

This paper describes Plural, a tool that automates this approach (section 3), and reports on a case study in using Plural on open-source code that uses Java APIs (section 4).

Copyright is held by the author/owner(s).

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

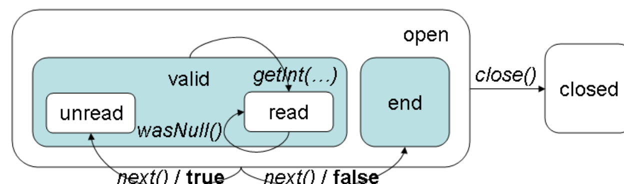

Figure 1. Simplified JDBC `ResultSet` protocol. Rounded rectangles denote states refining another state. Arches represent method invocations, optionally including return values.

2. Protocols with Access Permissions

In my approach, developers can associate objects with a *hierarchy* of typestates. For example, while a result set is *open*, it is convenient to distinguish whether the result currently points to a *valid* row or reached the *end* (figure 1).

Methods correspond to state transitions and are specified with *access permissions* that describe not only the state required and ensured by a method but also how the method will access the references passed into the method. We distinguish exclusive (unique), exclusive modifying (full), read-only (pure), immutable, and shared access. Furthermore, permissions include a *state guarantee*, a state that the method promises not to leave (Bierhoff and Aldrich, 2007). For example, `next` can promise not to leave *open* (figure 1).

Permissions can be *split* when aliases are introduced. For example, we can split a unique permission into a full and a pure permission to introduce a read-only alias. Using *fractions* (Boyland, 2003) we can also *merge* previously split permissions when aliases disappear (e.g., when a method returns). Fractions are conceptually rational numbers between zero and one. In previous work, fractions below one make objects immutable; in my approach, they can alternatively indicate shared modifying access. Splitting a permission into two means to replace it with two new permissions whose fractions sum up to the fractions in the permission being replaced. Merging two permissions does the opposite.

3. Plural: Access Permissions for Java

Plural implements the type system proposed in Bierhoff and Aldrich (2007) as a static dataflow analysis for Java. Developers use Java 5 annotations to specify method pre- and

```
public interface ResultSet {
  @Full(guarantee = "open")
  @TrueIndicates("unread")
  @FalseIndicates("end")
  boolean next();

  @Full(guarantee = "valid", ensures = "read")
  int getInt(int column);

  @Pure(guarantee = "read")
  boolean wasNull();

  @Full(ensures = "closed")
  void close(); ... }
```

Figure 2. Simplified `ResultSet` specification in Plural (using the typestates shown in figure 1).

post-conditions with access permissions (figure 2). No annotations are needed inside method bodies: Plural infers how permissions flow through the code—including loops—and checks that the declared post-condition holds. The analysis is modular because Plural "trusts" annotations on called methods and checks their bodies separately.[1]

The previously proposed type system "guesses" where permission splits and merges occur. Plural deterministically infers permission splits by collecting constraints on fractions. For every method call argument, Plural hypothesizes a split of the current permission. Constraints are added to make sure the fractions used in the new permissions sum up to the fractions in the permission being replaced. Additional constraints are added to ensure that the called method's pre-condition can be satisfied. Then, the post-condition is merged eagerly with permissions that remained with the caller. State guarantees are introduced and dropped lazily.[2]

4. Case Study: Apache Beehive

Beehive[3] is an open-source framework for declarative resource access. I have focused on the part of Beehive that accesses relational databases. I ran Plural on 10 Beehive source files with a total of ca. 1100 lines of code, which takes about 36 seconds on a 800MHz laptop.

I specified four Java standard APIs, highlighting Plural's ability to treat different APIs orthogonally. I can only discuss two of them here; the others (exceptions and regular expressions) are each only used once in Beehive.

JDBC. Java Database Connectivity defines interfaces for accessing relational databases. Roughly, `Statements` can send SQL commands on `Connections` to a database. Query results are represented as `ResultSets`. The three mentioned interfaces each define dozens of methods with about 20 lines of documentation each, for a total of more than 7000 lines of code. It took nearly three days to annotate these interfaces based on their documentation.

Figure 2 shows a small fragment of the `ResultSet` specification. Notice that `next`'s return value indicates if the result was advanced to a *valid* row or not. If the result points to a *valid* row then cell values can be read with `getInt` (and omitted similar methods). The only pure method `wasNull` tests if the last *read* value was `NULL`.

Iterators. Beehive implements an `Iterator` over a `ResultSet`. Plural can establish the connection between those two interfaces using *state invariants*, i.e., predicates over the underlying result set that specify iterator states. Interestingly, my vanilla iterator specification (see Bierhoff and Aldrich, 2007) assumes `hasNext`, which tests if an element can be retrieved, to be pure. Beehive's `hasNext`, however, is impure because it calls `next` on the `ResultSet`.

Observations. Plural's modular approach allowed me to move outwards from methods calling into JDBC interfaces to callers of those methods. It was often difficult to understand the design intent implicit in Beehive's code. Conversely, the JDBC interfaces were straightforward to specify from their extensive documentation.

Beehive is tricky to reason about because it aliases result sets through fields of various objects. In one place, the result set is implicitly passed in a method call. I turned it into an explicit method parameter; otherwise, Plural was able to analyze the code as-is. I assumed one class to be non-reentrant, but I believe with a more complicated specification the class can be analyzed assuming re-entrancy.

Acknowledgments

I thank Nels Beckman for his help with implementing Plural and Jonathan Aldrich for advising me in this research. This work was supported in part by the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems" and DARPA contract HR00110710019.

References

M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320. ACM Press, Oct. 2007.

K. Bierhoff and J. Aldrich. PLURAL: Checking protocol compliance under aliasing. In *ICSE-30 Companion*, pages 971–972. ACM Press, May 2008.

J. Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72. Springer, 2003.

R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490. Springer, 2004.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.

[1] For downloading Plural see the author's website.

[2] I recently demonstrated an early version of Plural that could not merge permissions or reason about object fields (Bierhoff and Aldrich, 2008).

[3] `http://beehive.apache.org/`