PLURAL: Checking Protocol Compliance under Aliasing

Kevin Bierhoff
Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
kevin.bierhoff @ cs.cmu.edu

Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich @ cs.cmu.edu

ABSTRACT

Enforcing compliance to API usage protocols is notoriously hard due to possible aliasing of objects through multiple references. In previous work we proposed a sound, modular approach to checking protocol compliance based on typestates that offers a great deal of flexibility in aliasing [1]. In our approach, API protocols are defined based on typestates. Every reference is associated with a *permission*, and reasoning about permissions is appropriately conservative for the "degree" of possible aliasing admitted by a permission.

This paper describes Plural, a tool to automatically enforce typestate-based protocols using permissions in Java. API developers can specify protocols with simple annotations on methods and method parameters. A static flow analysis tracks permissions in code that uses specified APIs and issues warnings for possible protocol violations.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented programming*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, Verification, Languages, Reliability

1. INTRODUCTION

Modern software development is highly reliant on re-use. The use of third-party 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 API implementations to work correctly. A usage protocol is concerned with temporal ordering of events: loosely speaking, they define legal sequences of method calls on objects. For example, a file object can be opened (once), then read multiple times, and finally closed, but it cannot be read after it was closed or before it was opened.

This paper presents Plural, an automated tool that gives developers comprehensive help in following API protocols statically, without executing their code. Plural uses *typestates* [4] to express protocols, such as the file protocol discussed above, as finite state machines. Plural is based on our previous work on using *permissions* [2] for sound (no errors missed) and modular (every method checked separately) protocol checking while allowing a great deal of flexibility in aliasing objects [1]. Our work is influenced by Fugue, the first typestate checker for an object-oriented language [3].

The following section summarizes permissions. Afterwards, we describe the functionality implemented in Plural and illustrate the tool with two examples protocols for input streams and iterators.

Copyright is held by the author/owner(s).
ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

2. PERMISSIONS

Permissions are associated with object references and govern how objects can be accessed through a given reference [1]. They can be seen as rely-guarantee contracts between the current reference and all other references to the same object: they provide guarantees about other references and restrict the current reference to not violate others' assumptions. Permissions capture three kinds of information:

1. *What kinds of references exist?* We distinguish read-only and modifying references, leading to the five different kinds of permissions shown in figure 1.

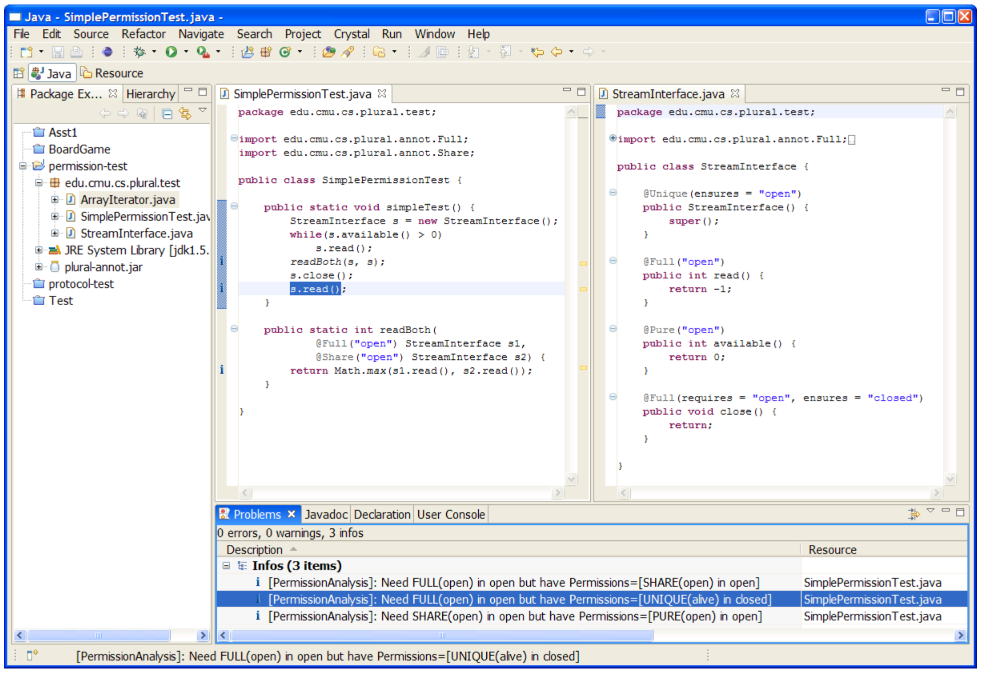
2. *What state is guaranteed?* A guaranteed state cannot be left by any reference. References can rely on the guaranteed state even if the referenced object was modified by other modifying references.

3. *What do we know about the current state of the object?* Every operation performed on the referenced object can change the object's state. In order to enforce protocols, we ultimately need to keep track of what state the referenced object is currently in.

Permissions can be *split* when aliases are introduced. For example, we can split the initial unique permission for an object (that is usually returned from a constructor call) into a full and a pure permission to introduce a read-only alias.

Access through	Current permission has ...	
other permissions	Read/write access	Read-only access
None	unique	–
Read-only	full	immutable
Read/write	share	pure

Figure 1: Access permission taxonomy



```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;
import edu.cmu.cs.plural.annot.Share;

public class SimplePermissionTest {

    public static void simpleTest() {
        StreamInterface s = new StreamInterface();
        while(s.available() > 0)
            s.read();
        readBoth(s, s);
        s.close();
        s.read();
    }

    public static int readBoth(
            @Full("open") StreamInterface s1,
            @Share("open") StreamInterface s2) {
        return Math.max(s1.read(), s2.read());
    }

}
```

```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;

public class StreamInterface {

    @Unique(ensures = "open")
    public StreamInterface() {
        super();
    }

    @Full("open")
    public int read() {
        return -1;
    }

    @Pure("open")
    public int available() {
        return 0;
    }

    @Full(requires = "open", ensures = "closed")
    public void close() {
        return;
    }

}
```

0 errors, 0 warnings, 3 infos

Description	Resource
Infos (3 items)	
[PermissionAnalysis]: Need FULL(open) in open but have Permissions=[SHARE(open) in open]	SimplePermissionTest.java
[PermissionAnalysis]: Need FULL(open) in open but have Permissions=[UNIQUE(alive) in closed]	SimplePermissionTest.java
[PermissionAnalysis]: Need SHARE(open) in open but have Permissions=[PURE(open) in open]	SimplePermissionTest.java

[PermissionAnalysis]: Need FULL(open) in open but have Permissions=[UNIQUE(alive) in closed]

Figure 2: Plural screenshot with stream protocol (right), stream client (left) and generated warnings (below).

3. PLURAL BY EXAMPLE

The Plural tool verifies protocol compliance based on permissions in Java. This goal motivates the tool's name: "Permissions Let Us Reason about ALiases".

Plural is implemented as a static flow analysis that checks conventional Java source code. For every program variable (the method receiver *this*, method parameters, and local variables), Plural determines the available permission at every program point. Every constructor or method call potentially affects the permissions associated with the program variables. These changes are governed by the specifications of the called constructors and methods.

We use JSR-175 annotations to specify protocols with permissions. Plural is a plug-in to the Eclipse IDE and indicates protocol violations using Eclipse's problems view (figure 2).

Input Stream Specification. Java input streams allow applications to read character sequences from various sources including files and devices. The input stream protocol can be modeled with two states *open* and *closed*. The `read` method can only be called on an *open* stream. In fact, `read` *guarantees* the the *open* state. The `close` method can also only be called in state *open*, but it transitions the stream to the *closed* state, forbidding further calls to either `read` or `close` (figure 2, right).

Checking a Stream Client. With this specification, Plural will flag protocol violations such as the ones displayed in figure 2 (bottom). (1) A share permission does not satisfy `read`'s specification which requires a full permission. (2) Calling `read` after closing *s* causes an error because the referenced stream is in the wrong state. (3) The unique permission available for *s* cannot be split into a full and a share permission (see section 2).

Branch-sensitivity for iterators. Iterators provide a method `next` to retrieve objects contained in a collection one by one. It is an error to call `next` if all objects were already retrieved. The *dynamic state test* method `hasNext` can be used to test if another object is available or not. In order to take such tests into account, Plural performs a *branch-sensitive* flow analysis: the analysis is aware of conditions known to be true inside of `if` statements or loops.

4. CONCLUSIONS

This paper presents Plural, a protocol conformance checking tool for protocols specified with typestates and permissions. Plural uses Java annotations to declare protocols and plugs into the Eclipse IDE. We show how Plural supports verifying clients of an input stream and an iterator interface.

Acknowledgements

This work was supported in part by NSF grant CCF-0546550, the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", DARPA contract HR00110710019, the U.S. Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

5. REFERENCES

[1] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 301–320, Oct. 2007.

[2] J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.

[3] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

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