Plural: A Typestate Checker Based on Access Permissions

Kevin Bierhoff Nels E. Beckman Paul Richardson Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
{kbierhof,nbeckman,aldrich}@cs.cmu.edu, psrichar@andrew.cmu.edu

ABSTRACT

In this paper we present Plural, a static, modular typestate checker for Java programs. Plural is interesting in that it is fully-featured; it includes a number of features that allow programmers to write expressive specifications, it includes inference mechanisms to help ease the specification burden, and it provides tools to help programmer understanding.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*class invariants, programming by contract*

General Terms

Verification

1. INTRODUCTION

In this paper we present Plural, a static typestate checker for the Java language. Plural is a mature and fully-featured tool; it provides numerous features for finding programmer mistakes, allowing more expressive specifications, is sound for all of Java and has special support for many of Java's more intricate features. Plural integrates into Eclipse's Java IDE and can perform typestate checking on entire projects in seconds. In this paper we will describe the primary features of Plural and how they can be used to more effectively check protocols in programs. While some of Plural's features are not novel, taken as a whole they can be seen as a contribution to the state-of-the-art in typestate checking. In short, we believe that Plural is the most mature and usable typestate checker currently available.

Plural is a typestate checker (which we will alternatively refer to as an *object protocol checker*). Object protocols are temporal restrictions on the order in which methods can be called and are part of the expected usage of certain classes. One classic example of an object protocol is Java's `FileReader` class in the java.io package. After construction, a file reader object is open and the `read` method can be called. Once the `close` method has been called on a file reader, subsequent calls to `read` will result in an `IOException` being thrown.

Object protocols are typically not checked by the compiler but rather are described in comments and documentation accompanying a class. For this reason considerable effort on the part of the programming languages and software engineering communities has gone into developing automated approaches for detecting or preventing object protocol violations [17, 11, 10, 15, 4, 1]. Plural is a static, modular object protocol checker for Java that implements the approach described by Bierhoff and Aldrich [4]. In this approach, programmers specify protocols using pre- and post-conditions and an alias-control mechanism called *access permissions*, which allow modular typestate verification in the face of a variety of aliasing patterns (see Section 2).

This paper presents the Plural tool, focusing on the features that make it easier or more pleasant for programmers to use. Some of these features promote expressiveness (e.g., method cases). However, we did not want expressiveness to come with a high annotation burden. Therefore, Plural has been designed to lower the specification burden (via inference), ensuring that programmers are never required to write specifications within a method body. Finally, a number of features help to make debugging and program understanding easier (e.g., the Fiddle visualization tool).

The Plural tool has been mentioned in earlier publications [1, 5]. However, neither of these works were able to devote much time to the tool itself. Here we will go into much more depth about the features of the tool, and cover many features that were not mentioned in earlier work.

This paper proceeds as follows: Section 2 gives a brief recap of the access permissions method of typestate checking. Section 3 describes how Java 5 annotations are used to write typestate specifications. Section 4 describes some of the more advanced specification features provided by Plural. Section 5 shows how Plural can perform inference in order to lessen the specification burden. Section 6 shows how Plural can be used to check some concrete program values. Section 7 describes Plural's features for reporting errors and supporting programmer understanding. Finally, Section 8 evaluates Plural's performance in practice.

2. ACCESS PERMISSIONS

Plural is a implementation of a flow-based type system first presented by Bierhoff and Aldrich [4], a methodology based upon the concept of *access permissions*. Briefly, we will recap the most important details of this methodology.

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

Figure 1: Access permission taxonomy

Access permissions are static predicates associated with program references by the type system. The access permission associated with a reference r holds the abstract state (e.g., Closed/Open) that the object referenced by r is known to inhabit at that particular program location. A typical problem with static, modular typestate checkers is that of aliasing. Without any aliasing knowledge, we would have to conservatively assume that other references to r exist, and that those references have been used to modify the state of the object after any method call. In order to get a handle on this problem, each access permission is one of five *permission kinds*, each recording whether or not the current context has the permission to access (read or modify) and whether or not other parts of the program have permissions. The five permission kinds are summarized in Figure 1.

Programmers write pre- and post-conditions which say what access permission must be available before a method is called, and what permission will be returned when the method returns. Since the access permission specifies an abstract state for the associated reference, the overall protocol for a class is encoded via pre- and post-conditions on its methods. Typestate checking proceeds on a per-method basis, assuming the permissions provided by the pre-condition, checking that enough permission is available for each method call, and finally, checking that the permission remaining at the end of the method body satisfies the post-condition.

Permissions are also notable in that they are based on fractional permissions [7]. This allows strong permissions like unique to be divided into multiple, weaker permissions like share and then later be recombined to form the original unique permission. Fractions help statically determine when all of the pieces of permission have been reclaimed.

Finally, access permissions also support state hierarchies and guaranteed states within those hierarchies. State hierarchies allow a class to define substates of existing states. For example, a stream class might have Open and Closed states. The Open state may be further refined into a Ready state, where further data can be received from the stream, and an EOF state, where nothing further can be received. State guarantees are a part of each access permission, and guarantee statically that no reference can be used to leave a particular state. For example, a share permission to a stream could guarantee it to be in the Open state. This will still allow multiple references to read from the stream, and yet provides the assurance that none of the references will be used to close the stream.

Now that we have seen a little bit about how access permissions work and how they can be used to check typestate, in the next section we will show how Java annotations can be used to write permission specifications and how we encode those permissions inside Plural.

3. PLURAL

```
1  @Full(requires="HasNext",ensures="alive")
2  @ResultShare
3  T next() { ... }
4
5  static int sizeOf(@Pure List<?> l) {
6    return l.size();
7  }
```

Figure 2: Basic specifications in Plural

Plural is a static modular typestate checker for Java programs. It is a flow and branch sensitive analysis that relies on a constraint-solver for the solutions to fractional constraints. Plural is built as a plugin to the Eclipse environment. Plural is an extension of Crystal,[1] an educational framework for developing dataflow analyses. It is open-source and is freely available for download.[2]

Specifications are written using Java annotations, and as a result, projects checked by Plural are syntactically legal Java programs. In this section we will explain how to write basic specifications in Plural.

3.1 Plural Specification Language

Plural uses Java 5 annotations in order to allow programmers to specify the access permissions required for each reference in a program. In this section we will describe those annotations and show how permissions are encoded in our analysis.

We have attempted to design the specifications so that they are as easy to use as possible. For each of the five permission kinds, there are two annotations, one annotation for specifying method receivers and parameters, and the other for specifying return values. Figure 2 shows three of these basic annotations.

The `next` method is a method of the `Iterator` interface. Its specification reads as follows: In order to call this method, the caller must have a full permission (exclusive modifying reference to a possibly-aliased object) to the iterator itself. That iterator must be in the HasNext state. When the method returns, that same full permission will be given back to the caller, and the iterator will be in the alive state. (In our system of state hierarchies, alive is the root state implicitly defined for every object, so this is another way of saying the iterator will be in an unknown state.) The method returns a share permission to the object that it returns, and that object will be in the alive state. "alive" is the default for all state specifications so any time a state specification is omitted, as in `@ResultShare`, alive is automatically used.

The static `sizeOf` method requires a pure permission to its sole parameter, which will allow it to call read-only methods on `l`. This permission will be returned when the method returns. The annotation has no further specification and so defaults are used; an argument must be in the alive state to call the method and will be returned in the alive state.

Figure 3 shows how programmers declare states in the first place, and how to form a state hierarchy. The `@States` annotation allows programmers to introduce a collection of new, mutually exclusive abstract states for a class or interface. Those new states *must* refine an existing state. In

[1] `http://code.google.com/p/crystalsaf/`
[2] `http://code.google.com/p/pluralism/`

```
1  @Refine({
2    @States({"Open","Closed"}),
3    @States(refined="Open",
4            value={"Ready","EOF"}) })
5  class Stream { ... }
6
7  void read(@Share("Open") Stream s) {
8    while(s.hasMore()) { s.read(); }
9  }
```

Figure 3: Declaring a state hierarchy for our stream

the case of Ready and EOF, the Open state is refined (in other words, Ready and EOF are substates of Open). Open and Closed, however, are not specified as being substates of any state. They therefore implicitly refine alive, the default. The `read` method requires a *guaranteed* permission to its parameter `s`. Its specification says; in order to call the `read` method, the caller must have a share permission to the stream (an aliased, non-exclusive modifying permission). This permission must guarantee that `s` will not used to leave the Open state (i.e., that it will not be closed). This prevents us from calling `close` on `s` but it also ensures us that no other reference to `s` can be used to call `close`.

State Invariants. Our approach not only allows us to check if clients of an API respect its protocols, it can also check that the API implements the protocol correctly. The key abstraction for this are *state invariants*, which we adapted from Fugue [11]. A state invariant associates an abstract state of a class with a predicate over the fields of that class. Generally this predicate consists of access permissions for fields of that class. State invariants are specified at the class level, using the `@Invariants` and `@State` annotations. In the following example, a network stream is specified as being open whenever its underlying socket is also open:

```
1  @Invariants({@State(name="Open",
2    inv="unique(s) in Open",
3   @State(name="Closed",
4    inv="unique(s) in Closed"})
5  class NetStream extends Stream {Socket s; ... }
```

Permission Representation. So how is a permission like `@Unique` represented internally? Permissions in Plural are represented by a fraction function, a map from states in the state hierarchy (from alive to the guaranteed state) to fractions between zero and one, inclusive. A separate fraction tracks what permission is available to modify the state below the guarantee, and a flag helps separate share permissions from immutable permissions, which are otherwise identical. (The immutable permission was not a part of the original theory presented by Bierhoff and Aldrich [4].) Consider a unique permission to an open stream, *s*. Plural represents such a permission in the following manner:

$$\{\texttt{map} : [\mathsf{alive} \mapsto 1], \texttt{below} : 1, \texttt{state} : \mathsf{Open}, \texttt{imm} : \texttt{false}\}$$

We could then split that permission into two share permissions with the guaranteed state Open. After establishing that the stream has data to read, we would have the following permission:

$$\{\texttt{map} : [\mathsf{alive} \mapsto \tfrac{1}{2}; \mathsf{Open} \mapsto \tfrac{1}{2}], \texttt{below} : \tfrac{1}{2}, \texttt{state} : \mathsf{Ready}, \texttt{imm} : \texttt{false}\}$$

In this section we showed a few of the most commonly-used specifications. In the next section we will describe some of the more interesting ones.

4. ADVANCED SPECIFICATIONS

The specifications described in this section allow for quite a bit of additional flexibility on the part of the programmer. Here we describe borrowing, method cases, marker states, dependent objects and a variety of features specific to Java.

4.1 Borrowing

Plural has support for permission *borrowing* [9]. A borrowed permission is a permission that is temporarily lent from the context of one method body to a method called within the body. In other words, methods that return exactly the same permission that they are given. In our case studies we found that permission borrowing is the most common form of permission specification, and so we made it the default. To specify a borrowed permission, a programmer writes one of the five standard permission annotations and sets the `returned` flag to 'true.' For example, the following specification describes a borrowed permission:

```
1  void read(@Share(returned="true",
2    guarantee="Open") Stream s);
```

This `read` method indicates that callers will get back the exact same share permission, meaning the same fractions, as the one they passed in. Since 'true' is the default value for the `returned` property, this specification is semantically identical to the one on line 7 of Figure 3.

Support for borrowing increases both precision and performance. To see how precision is increased, consider the following: When a method call requires introducing a state guarantee for a permission that is borrowed, then it is *always* safe to drop the state guarantee after the call. Because the calling context knows it is getting back the exact same fraction it lent out, it can be sure that no other permission was taken off of the permission, and therefore no new permissions can be relying on the guarantee.

Support for borrowing also increases analysis performance because we do not have to keep constraints introduced for borrowed permissions beyond the current call site and can instead revert to the permission from before the call. In practice, this cuts the time it takes our regression suite to run roughly in half!

One alternative to permission borrowing is *consumption*. By setting the `returned` property to 'false,' a specification indicates that it will not return the permission it is given. This usually occurs when a method intends to store some permission in an object field. Finally, the `@Perm` annotation can be used to return a permission of a different fractional amount, or even of an entirely different permission kind. The following specification describes a method that requires a full permission and gives back a pure permission:

```
1  @Perm(requires="full(s) in Open",
2        ensures="pure(s) in Open")
3  void read(Stream s);
```

4.2 Method Cases

The idea of method cases goes back to behavioral specification methods, e.g., in the JML [16]. Method cases amount to specifying the same method with multiple pre-/post-condition

pairs, allowing methods to behave differently in different situations. We early on recognized their relevance for specifying API protocols [3, 2], but we are not aware of any other protocol checking approaches that support method cases.

In Plural, programmers can specify a method with multiple cases using the `@Cases` annotation, which holds an array of `@Perm` annotations, one for each case. A method with multiple specification cases is checked under each case, verifying that, given the pre-condition the method satisfies the post-condition when it returns. Section 5.2 describes the inference mechanism that is used to choose the correct case at a given call site.

4.3 Marker States

Plural can treat states special that are fixed throughout the object lifetime. We call these *marker states*, which are reminiscent of (flow-insensitive) type qualifiers [13] and type refinements [12]. Knowledge about an object being in a marker state, once gained, cannot be lost, which can simplify checking API clients. Marker states are also interesting semantically as they indicate object properties that are fixed at construction time, thereby directly refining conventional Java types with additional, flow-insensitive information that does not change throughout the object's lifetime.

Notably, marker states can capture the well-known distinction between "mutable" and "unmodifiable" collections as they are defined in the Java Collections Framework. We can also use marker states for distinguishing "readonly" and "modifying" iterators over mutable collections [2].

4.4 Dependent Objects

Another feature of many APIs is that objects can become invalid while other, related objects are manipulated in certain ways. As an example, consider the JDBC library for connecting to SQL databases [5]. A query result, encapsulated in an instance of `ResultSet`, will become invalid if the originating database connection is closed. (A similar problem, called concurrent modification, exists with iterators [2].) There are no other automated modular protocol checkers that we know of that can handle these protocols, although recent global protocol checking approaches can [6].

Our approach is to "capture" a permission to the depended-upon object (the connection in the example) in the dependent object (e.g., result set). The captured permission prevents the problematic operation (e.g., closing the connection) from happening. When a dependent object is no longer using a captured permission, that permission can be "released", re-enabling operations on the depended-upon object. Captured permissions are declared in Plural with a `@Param` annotation, and methods annotated with `@Release` explicitly release permissions.

Frequently, however, such releasing methods do not exist, or their use is optional. In this case, our solution is to employ a live variable analysis to detect *dead objects*, i.e., dead references to objects with unique permissions, and release any captured permissions from these dead objects. Others have modeled these protocols with linear implications [8, 14] but it is unclear how well those approaches can be automated. In contrast, Plural can easily detect dead objects based on permission and live variable information.

4.5 Dealing with Java

This section details how Plural handles Java features such as constructors, static fields, and arrays. All of the features discussed in this section are common in practical programming languages; therefore, the discussion applies to languages other than Java as well.

Inheritance. We mentioned previously that Plural allows associating abstract states with invariants over instance fields, and that unpacking and packing "trades" between a receiver permission in a given state and permissions to access the receiver's fields according to that state's invariant. Subclasses may define their own state invariants, and the superclass can be in a different state from its subclass. There are some situations we have found where this is necessary. For example, Bierhoff and Aldrich [3] described a buffered reader class that, in its constructor, opens its superclass (a standard reader), reads all data into a buffer, and then closes the superclass. The subclass itself is in the `Open` state, and its clients see the reader as open, however every object in the hierarchy is *not* required to be open.

In order to support this idiom, we start by hypothesizing that the permission to the fields of each class in the hierarchy are grouped, in what we call a *frame* [11]. In practice, the specifications in each class refer to either the *frame permission*, which describes the state of current group of fields in the hierarchy, or the *virtual permission*, which describes the entire object as it is seen from the client's perspective. The virtual permission is always the same state as the frame permission of the actual instantiated class. When a class defines its state invariants, it can specify the required state of the superclass in each invariant using a permission to the frame of the superclass.

Receiver permissions have to be explicitly marked as frame permissions in Plural using the `use` attribute (virtual is the default), and they are tracked separately when checking method implementations. Plural allows only frame permissions to be unpacked and uses only virtual permissions to satisfy pre-conditions of virtual methods [4]. If a specification requires both the frame and the virtual permission for its receiver, then Plural will check the method twice. In the first case, it assumes that the current class *is* at the bottom of the class hierarchy, and therefore treats the virtual and frame permissions as being the same. In the second case, it assumes that the current class has some subclass, and therefore the frame and virtual permissions are treated separately. (The first or second check is omitted for `abstract` and `final` classes, respectively.)

On the client-side, all receiver permissions are exposed as regular (virtual) permissions, making the distinction irrelevant. This is possible because our theory requires overriding methods with frame permissions, allowing virtual permissions to be soundly coerced into frame permissions during dynamic dispatch [4].

Constructors. We leverage API implementation checking to reason about constructors by injecting an unpacked unique *frame* permission (see above) for the receiver into the initial context. The constructor post-condition is developer-declared and checked as usual, and calls to super-constructors can be handled like calls to inherited methods.

If virtual method calls need to be made from within a constructor, then permission to the virtual frame of the newly constructed object must be specified as part of the pre-condition. Subclasses that use the constructor will be

required to satisfy the pre-condition for the virtual frame. This has the beneficial side-effect of alerting subclasses to the possibility of a virtual call, which might dispatch to an overridden method. For soundness, the body of a constructor requiring virtual permission is verified twice: once assuming the constructor is being invoked from a subclass and therefore has a virtual permission, and once assuming the constructor is the one for the object type being instantiated. In the later case, a frame permission can be used to satisfy any method call requiring a virtual permission.

Private Methods. Private methods can by definition not be overridden. This means that the *frame* they work on is statically known to be the class they are defined in. Therefore, Plural handles calls to private methods akin to *super*-calls: frame permissions required by a private method can be satisfied with frame permissions available at the call site. This simplifies specifying the methods that call private methods because they typically do not need to require both a frame (for their own field accesses) and a virtual permission (for calling the private method) but just a frame permission.

Static Fields (Globals) . Sometimes we need to associate permissions with static fields. (Static fields are similar to global variables in procedural languages.) Plural currently allows doing so with permission annotations directly on the field. To simplify matters, however, static fields can only be associated with permissions that can be duplicated (more precisely, split into equally powerful permissions, i.e. share, immutable, and pure). This avoids problems when permissions from a static field access are still in use when the same static field is accessed a second time.

In order to allow unique and full permissions for static fields, one could treat static fields as fields of the surrounding "class" object, which would require passing around permissions for the surrounding "class" objects to where their static fields are accessed. Another option would be a simple effect system that indicates which static fields are accessed in the dynamic scope of a method.

Arrays and Collections. Currently Plural associates arrays with a permission of their own and ensures that a modifying permission (unique, full, or share) is available upon stores into the array. It does not, however, perform any kind of tracking for the array elements. No permission can be stored by writing to an array and no permission can be retrieved by reading from one. While this is sound, we would like to implement better support for arrays in the future. However, Plural *does* allow for the specification of classes that are polymorphic in the permissions that they can hold. This allows, for instance, a programmer to specify a stack class that can hold elements of any permission kind. When instantiated at a particular permission kind, the stack will only accept and return permissions of that kind.

5. SPECIFICATION INFERENCE

In this section we describe various features of Plural that ease the specification burden by performing some kind of inference. Our goal when designing Plural was to require a system in which programmers did not have to write any specifications within a method body, only at method and class boundaries. In the end, this was indeed the case.

5.1 Local Must-Alias Analysis

While permission annotations provide modular alias information at method boundaries, Plural can infer local aliasing. This is useful because programmers will occasionally assign a variable to a local variable, and we would like to allow them to do so without writing a separate permission specification for the local variable. In order to do so, Plural includes a simple must-alias analysis. In this analysis, each variable is associated with an abstract location. When the analysis encounters a variable copy instruction, it records this by setting variables to point to the same abstract location.

Plural generates fresh locations for references returned from method calls. These references may at run-time point to the same object as another local variable. In other words, local variables may alias even though our local alias analysis does not indicate this possibility. The use of permissions prevents problems in this case because Plural will associate different sets of permissions with the two locations which correspond to the same runtime object.

5.2 Invariant Checking

As described in Section 3, programmers can specify state invariants, predicates that must be true when an object is in a certain state. During the verification of these invariants, Plural performs packing/unpacking inference.

Whenever a receiver field is used, Plural *unpacks* a permission to the surrounding object to gain access to its state invariants [11, 4]. Essentially, unpacking means replacing the receiver permission with permissions for the receiver's fields as specified in state invariants. Before method calls, and before the analyzed method returns, Plural *packs* the receiver, possibly to a different state, by splitting off that state's invariant from the available field permissions [4].

In the formal presentation of this analysis, programmers were required to specify exactly which states they wanted to pack to (and similarly, up to what state in the hierarchy they wanted to unpack from). This would require extra permission annotations within the method body. One of our goals was to avoid annotations within method bodies. Therefore, Plural infers which state the receiver should be packed to, and from which state the receiver should be unpacked, essentially by trying them all. Whenever the receiver must be (un)packed, Plural attempts to (un)pack the receiver from or to all of the states defined by the object's type. Any states that it cannot (un)pack from or to will immediately be discarded. However, in the event that the state invariants of multiple states can be satisfied, Plural will duplicate the current context, making a copy for each successful state, and propagate each context forward in the analysis. Later on in the method body, if one of the contexts turns out to be unsatisfiable (presumably because (un)packing from or to that state was the wrong choice) it will be discarded and the other, satisfiable contexts will continue to more forward in the method.

5.3 Other Inference

In Section 4 we described multi-case method specifications. Calls to such methods are checked in a manner that is quite similar to packing and unpacking inference. Whenever a method is called that has multiple specifications, instead of requiring the programmer to say which specification he wants to use, Plural simply tries all of them, propagating forward the results of the cases whose pre-conditions were

satisfied, and dropping those that were not.

Because Plural is implemented as a dataflow analysis, and as such, can infer loop invariants. As multiple paths around a loop converge at the end of the loop, the contexts from each path are joined in a sound manner. This joining process roughly involves moving up the state hierarchy when two permissions to the same object in different states are joined, and to the weaker permission, when two permissions disagree as to what permission kind a reference has. Our lattice is of finite height, so this process is guaranteed to terminate. Unfortunately, the details of the lattice and operators on the lattice is somewhat involved, and is outside the scope of this paper. For details, see Bierhoff [2].

6. CONCRETE PREDICATES

Plural allows programmers to write specifications that depend on actual program values, such as booleans, in addition to specifications that depend on abstract states. This is necessary in order to successfully specify the use and implementation of protocols like that of the `Iterator` interface which allow what we call *dynamic state tests*. Dynamic state tests are runtime tests of the abstract state of an object. In the `Iterator` interface, the return value of the `hasNext` interface indicates whether or not there are further items available in the iterator, and therefore it is safe to call the `next` method. This section describes the mechanisms of Plural which enable the specification and verification of such protocols.

6.1 Concrete Values

Plural contains a sub-analysis that tracks concrete values for many variables. For variables of type boolean, this analysis tracks whether the variable is known to be true or false at each program point. For variables that are subtypes of Object, we track whether or not the variable is known to be null or non-null. This analysis is branch sensitive and therefore any conditional statements that check the truth or non-nullness of variables can provide our analysis with more precise information.

This information is useful, because we allow programmers to write specifications that mention these concrete values. Programmers can specify, for example, that a particular field of an object must be true whenever it the object is in a certain state. Consider the following specification for the `Blocking_queue` class (Section 8):

```
1  @States({"Open","Closed"})
2  @Invariants({
3   @State(name="Open",inv="closed==false"),
4   @State(name="Closed",inv="closed==true")})
5  class Blocking_queue {
6    boolean closed = false; ... }
```

This specification says that, whenever the `closed` field is true the queue will be in the Closed state, and otherwise it will be open. The information about concrete values provided by this analysis is used whenever an object is packed or unpacked. When packing to the Open state, for example, Plural will make sure that `closed` is known to be false. When unpacking an open queue, Plural tells the sub-analysis that the `closed` field is known to be false. Plural does not currently track integer values or any concrete values other than boolean values and nullness. We believe it could be suitably extended in this manner if desired.

```
1  // @States ... other specs
2  class Blocking_queue {
3    ...
4    @TrueIndicates("Closed")
5    @FalseIndicatse("Open")
6    boolean isClosed() {
7      return closed;
8    }
9  }
10
11 // Elsewhere...
12 if( blocking_queue.isClosed() ) {
13   return blocking_queue.dequeue(); // Error
14 }
```

Figure 4: Specification and use of a dynamic state test method

6.2 Dynamic State Tests

The same infrastructure that is used to track the concrete values of variables is also useful for checking dynamic states tests. Dynamic state tests are used in classes where the abstract state of an object can be queried at runtime. The `Iterator` interface and the `Blocking_queue` class from the previous section are both examples of this sort of protocol.

Let us describe the specification of dynamic state tests and their verification, both on the client-side and the implementation side. Consider the `isClosed` method of the blocking queue class shown in Figure 4. The `@TrueIndicates` annotation says that if the `isClosed` method returns true, then the queue will be in the closed state. Likewise, the `@FalseIndicates` annotations says that if the method returns false, the queue is open.

On the client-side, Plural performs a *branch-sensitive* flow analysis: if the code tests the state of an object, for instance with an `if` statement, then the analysis updates the state of the object being tested *according to the test's result*. In this case, Plural updates the queue's state to Open at the beginning of the 'then' branch in Figure 4 (line 13). Recent whole-program protocol analyses [6] would miss this error because they merely ensure that dynamic state tests are *called*.

On the implementation-side, these specifications are checked to ensure that the receiver object is actually in the state indicated by the return value. It checks at every method exit that, assuming `true` (or `false`) is returned, the receiver is in the state indicated by `true` (resp. `false`).

7. ERROR REPORTING

Since Plural specifications can occasionally be complex, we have developed a number of tools for error reporting and program understanding. In this section we discuss error reporting, the state diagram visualization tool, Fiddle, and the verification context viewer.

7.1 Error Reporting

Plural reports possible protocol violations at the point in the program where errors are detected. In particular, if a method pre-condition cannot be satisfied at a call site then Plural will issue a warning at that call site, explaining the nature of the violation. Plural finds violations by querying the results of the dataflow analysis and checks at each

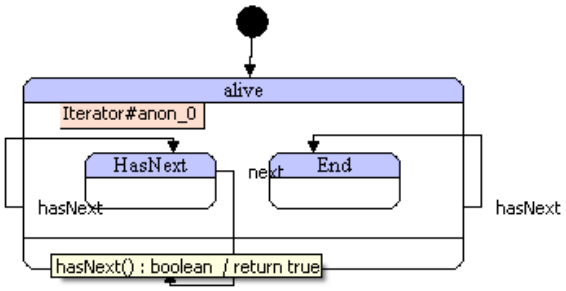

Figure 5: A state chart modeling the Iterator protocol as rendered by the Fiddle visualization plugin.

method call site whether the pre-condition is satisfiable.

7.2 Fiddle Visualization Tool

As an aid to programmer understanding, the Plural tool can visualize the protocols of classes and interfaces that define them. This functionality was designed as a separate Eclipse plugin, which we call *Fiddle*. Fiddle's role is fairly straightforward. It attempts to model the protocol of the currently selected type as a UML statechart diagram. When a programmer opens up a file in Eclipse, or when he selects a class or interface in Eclipse's outline or tree views, Fiddle draws a rendering of the protocol defined by that type. Plural maintains information about each type describing its state hierarchy, its methods, and which states those methods transition to and from. Fiddle simply depicts this information graphically. Programmers can hover over each state transition in order to get more information about the method's specification. Figure 5 shows a screenshot from Fiddle which contains a state chart representing the protocol of the `Iterator` interface.

7.3 Context Tree View

In practice, the inference performed by Plural in Section 5 can occasionally lead to unexpected output from the tool. Consider packing and unpacking inference; if at some point during a method body, the analysis cannot pack the receiver to a state that the programmer expects the receiver to pack to, but can pack to some other unexpected state, any subsequent errors reported by the tool can be quite surprising. The context may contain permission that the programmer does not expect, or may not contain permission that the programmer does expect. Such a situation has occurred during our case studies enough times to warrant more detailed output. Thus, the Plural tool provides what we call the Context Tree View.

The context tree view is a separate window inside Plural that allows programmers to view the state of the context at each program point. This view is structured as a tree. This is necessary because at certain points in the method body multiple contexts are introduced as the result of inference, and each context proceeds in its own manner as it steps through the remainder of the method. In Section 5 we described how, if the receiver could be successfully packed to multiple states at a particular program point, the context would be cloned and each of the contexts would be packed to a different state, and the entire collection of contexts would be propagated forward. In the context tree view, at any point where the current context was split into multiple contexts, the view represents this split as a branch, and tells the user why the split occurred. So, if the user wants to see why a particular method does not verify, he can follow the context from line to line, examining the particular packs, unpacks and method cases he expects should work, and eventually can see at which point that context was no longer satisfiable. In practice, this makes diagnosing reported errors much easier.

8. EXPERIENCE

We have used Plural, the typestate verification tool described in this paper to specify and verify a number of real programs. In this section, we show that the performance of Plural and the fraction inference algorithm it implements is usable in practice.

Figure 6 shows the results of running Plural on several programs. Because our analysis is a modular one, we have highlighted the number of methods in each program and the average time Plural spent on each method. As long as Plural shows reasonable performance on larger and more interesting methods, we expect our analysis will scale to programs with a very large total size.

In order to gauge the performance of Plural, we ran Plural on several programs, case studies and benchmarks and then recorded the time it took Plural to verify each program as well as some basic metrics about each program.[3] Figure 6 shows for each program (from left to right), the mean runtime of four runs in seconds, the total number of methods, the runtime per method in milliseconds, the number of lines of source code inside method bodies (excluding e.g., field initializers), the lines of source per method and the size in lines of source of the largest method. (For technical reasons, we could not determine the analysis time for the largest method separately.) We benchmarked five Java programs:

PMD. PMD[4] is a static analysis framework for Java that we specified and verified as part of an earlier case study [5]. This program is interesting because it uses the Java Iterator interface extensively, and it is relatively large. Plural successfully verified that the iterator protocol was obeyed. While the overall runtime was larger for PMD due to its size, the time spent verifying each method was rather low due to the relatively small number of permissions that were tracked per method.

Regression Suite. We ran Plural over the entire Plural regression suite, which consists of 132 classes and is meant both to test basic functionality as well as some more interesting cases that previously caused our analysis to exhibit bugs.

4InALine. 4InALine[5] is a Swing video game based on the board game Connect Four. As part of another case study we specified just the aliasing behavior of 23 of the classes in this program and used Plural to verify the consistency of those annotations.

[3]All programs were run on a Dell PC running Windows XP Service Pack 2, with a 3.2GHz Intel Pentium 4 and 2GB of RAM.
[4]`http://pmd.sourceforge.net/`
[5]`http://code.google.com/p/fourinaline/`

Program	Runtime (s)	Methods	Runtime / Method (ms)	LOC	LOC / Method	LOC in Largest Method
PMD	259.978	4198	61.93	30594	7.288	517
Regression Suite	10.151	491	20.67	1867	3.802	102
4InALine	6.575	206	31.92	1405	6.82	71
Beehive	9.758	52	187.65	623	11.98	114
Blocking_queue	3.278	25	131.10	142	5.68	43
Total	289.739	4972	58.27	34631	6.961	517

Figure 6: Results of running the Plural typestate checker on various programs.

Beehive. Apache Beehive[6] is a application framework that, among other things, simplifies the development of JDBC-based applications. As part of an earlier case study [5] we specified protocols in the Java collections framework, the JDBC framework, the Java regular expressions classes and the Throwable class (which defines a very simple protocol for the setting of its "cause"). We verified 12 classes in Beehive against these protocols.

Blocking_queue. Blocking_queue is a case study used to evaluate the NIMBY [1] analysis program, an extension to Plural which also checks correct usage of concurrency. This case study includes a "blocking queue" class, which defines an open/closed protocol, and several clients which use the queue in interesting ways.

Discussion. The performance of Plural was generally good. As we expected, there seems to be correlation between the size of the methods in a program and the amount of time it takes to analyze each method. The exception here is the Blocking_queue benchmark which takes longer to analyze than its average method size would suggest. We believe this is due to the relative complexity of the invariants and the large number of permissions that are being used throughout this program as well as the additional checks that must be performed in order to guarantee thread safety.

References

[1] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of Atomic blocks and typestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 227–244, Oct. 2008.

[2] K. Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, April 2009.

[3] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 217–226, Sept. 2005.

[4] 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.

[5] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, July 2009.

[6] http://beehive.apache.org/

[6] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *ACM Symposium on the Foundations of Software Engineering*, pages 36–47, Nov. 2008.

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

[8] J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *ACM Symposium on Principles of Programming Languages*, pages 283–295, Jan. 2005.

[9] E. C. Chan, J. T. Boyland, and W. L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *20th International Conference on Software Engineering*, pages 167–176, Los Alamitos, CA, Apr. 1998. IEEE.

[10] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

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

[12] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *ACM Symposium on Principles of Programming Languages*, pages 281–292, 2004.

[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[14] C. Haack and C. Hurlin. Resource usage protocols for iterators. In *International Workshop on Aliasing, Confinement and Ownership*, July 2008.

[15] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[17] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.