Practical API Protocol Checking with Access Permissions

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich

Institute for Software Research, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{kevin.bierhoff, nbeckman, jonathan.aldrich}@cs.cmu.edu

Abstract. Reusable APIs often define usage protocols. We previously developed a sound modular type system that checks compliance with typestate-based protocols while affording a great deal of aliasing flexibility. We also developed Plural, a prototype tool that embodies our approach as an automated static analysis and includes several extensions we found useful in practice. This paper evaluates our approach along the following dimensions: (1) We report on experience in specifying relevant usage rules for a large Java standard API with our approach. We also specify several other Java APIs and identify recurring patterns. (2) We summarize two case studies in verifying third-party open-source code bases with few false positives using our tool. We discuss how tool shortcomings can be addressed either with code refactorings or extensions to the tool itself. These results indicate that our approach can be used to specify and enforce real API protocols in practice.

1 Introduction

Reusable APIs often define usage protocols. Loosely speaking, usage protocols are constraints on the order in which events are allowed to occur. For example, a database connection can only be used to execute SQL commands until it is closed. It has been a long-standing research challenge to ensure *statically* (before a program ever runs) that API protocols are followed in client programs using an API. An often overlooked but related problem is ensuring that the protocol being checked is consistent with the actual implementation of that protocol. Both of these challenges are complicated by object *aliasing* (objects being referenced and possibly updated from multiple places)—the hallmark feature of imperative languages like C and Java.

We previously developed a *sound* (no false negatives) and *modular* (each method checked separately) type system that checks compliance with typestate-based protocols while affording a great deal of aliasing flexibility [7, 3]. *Typestates* [33] allow specifying usage protocols as finite-state machines. Our approach tracks *access permissions*, which combine typestate and aliasing information, and was proven sound for core single-threaded [7] and multi-threaded [3] object-oriented calculi. Unlike previous approaches, access permissions do *not* require

precise tracking of all object aliases (e.g. [15, 17]) or impose an ownership discipline on the heap (e.g. [2]).

We have implemented Plural, a tool that embodies our approach as an automated static analysis [9] and includes several extensions found to be useful in practice (section 2.2).[1] Plural requires developer-provided annotations on methods and classes, specifically on the APIs to be checked, although others may be necessary.

While previous modular protocol checking approaches have been proven sound and shown to work on well-known examples such as file access protocols, these approaches generally have not been evaluated on real APIs or third-party code bases. (Notable exceptions include Vault [15] and Fugue [17, 16]). The general lack of evaluation begs important questions: Can these approaches accurately specify protocols that occur in practice? Can they verify their use and implementation without an unreasonable number of false-positives? And can they do so at a low computational cost and without imposing too great of an annotation burden?

In this paper we attempt to answer these questions. Contributions of this paper include the following:

- **Specification.** We report on experience in specifying relevant usage rules for 440 methods defined in the Java Database Connectivity (JDBC) API for relational database access with our approach (section 3).[2] To our knowledge, this is the largest case study available in the literature that evaluates the applicability of a usage protocol specification method for real APIs. Several other Java APIs were specified and are also discussed.
- **Checking.** We summarize two case studies in using Plural on third-party open-source code bases.
  - We checked about 2,000 lines taken from the Apache Beehive project against the specified APIs (section 4).
  - We also checked PMD, a program of about 40KLOC, for compliance to a simple iterator protocol (section 5).

  We find that the code can be checked with few false positives and report the annotation overhead of using our tool. We also discuss how tool shortcomings can be addressed (section 6). To our knowledge, precision and annotation overhead measurements are not available for previous modular protocol checking approaches.
- **API Patterns.** We comment on several recurring patterns that we found to be interesting. These patterns represent challenges that any practical protocol enforcement technique should be able to handle. Moreover, several of these patterns are handled elegantly by the novel technical features of our system. For example:
  - It is crucial to track what we refer to as "dynamic state tests," methods that can query the abstract state of an object at run-time. For example, the `hasNext` method of the `Iterator` interface.

[1] Plural is open-source: `http://code.google.com/p/pluralism/`.

[2] API specifications are available at `http://www.cs.cmu.edu/~kbierhof/`.

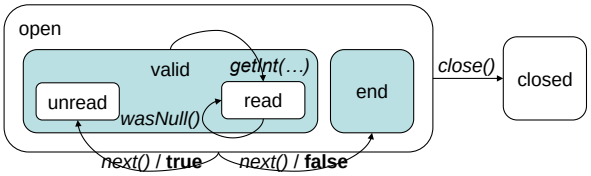

Fig. 1. Simplified JDBC `ResultSet` protocol. Rounded rectangles denote states refining another state. Arches represent method calls, optionally with return values.

- State guarantees, which allow multiple references to modify an object while all depending on the fact that it will not leave a particular state, were similarly quite useful.
- Inter-object dependencies, where the state of one object depends on the state of another, are common but can be precisely checked with our approach.

We first describe permissions and the Plural tool in section 2, and then we discuss our case studies in the subsequent sections. Section 7 summarizes related work and section 8 concludes.

2 Typestate protocols with access permissions

This section summarizes our previous work on access permissions [7] for enforcing typestate protocols and our work on Plural, an automated tool for checking permission-based typestate protocols in Java [9]. Plural is described in more detail in the first author's dissertation [5].

2.1 Access Permissions

Our static, modular approach to checking API protocols is based on access permissions, predicates associated with program references descring the abstract state of that reference and the ways in which is may be aliased.

In our approach, developpers start by specifying their protocol. Figure 1 shows a simplified protocol for the JDBC `ResultSet` interface as a Statechart [22]. `ResultSet`s represent SQL query results, and we will use their protocol as a running example in this and the following section.

We allow developers to associate objects with a *hierarchy* of typestates, similar to Statecharts [22]. For example, while a result set is *open*, it is convenient to distinguish whether it 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

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

Table 1. Access permission taxonomy

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 (table 1). Furthermore, permissions include a *state guarantee*, a state that the method promises not to leave [7]. For example, `next` can promise not to leave *open* (figure 1).

Permissions are associated with object references and govern how objects can be accessed through a given reference [7]. 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 table 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 only co-exist if they do not violate each other's assumptions. Thus, the following aliasing situations can occur for a given object: a single reference (unique), a distinguished writer reference (full) with many readers (pure), many writers (share) and many readers (pure), and no writers and only readers (immutable and pure).

Permissions are linear in order to preserve this invariant. But unlike linear type systems [34], they allow aliasing. This is because permissions can be *split* when aliases are introduced. For example, we can split a unique permission into a full and a pure permission, written $\mathsf{unique} \Rrightarrow \mathsf{full} \otimes \mathsf{pure}$ to introduce a read-only alias. Using *fractions* [11] we can also *merge* previously split permissions when aliases disappear (e.g., when a method returns). This allows recovering a more powerful permission. For example, $\mathsf{full} \Rrightarrow \frac{1}{2} \cdot \mathsf{share} \otimes \frac{1}{2} \cdot \mathsf{share} \Rrightarrow \mathsf{full}$.

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

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

    @Pure(guarantee = "valid", requires = "read")
    boolean wasNull();

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

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

2.2 Plural: Access permissions for Java

Our tool, Plural, is a plug-in to the Eclipse IDE that implements the previously developed type system [7, 3] as a static dataflow analysis for Java [9].

In the remainder of this section we show example annotations and explain how permissions are tracked and API implementations are verified. Then we discuss tool features we found useful in practice.

Developer annotations. Developers use Java 5 annotations to specify method pre- and post-conditions with access permissions (figure 2). Figure 2 shows a simplified `ResultSet` specification with Plural's annotations (compare to figure 1). Annotations on methods specify *borrowed* permissions for the receiver, while annotations on method parameters do the same for the associated parameter. Borrowed permissions are returned to the caller when the method returns. The attribute "guarantee" specifies a state that cannot be left while the method executes. For example, `next` advances to the next row in the query result, guaranteeing the result set to remain *open*. Cell values can be read with `getInt` (and similar but omitted methods) if the result points to a *valid* row. Conversely, a required (or ensured) state only has to hold when the method is called (or returns). For instance, only after calling `getInt` is it legal to call `wasNull`. Additional annotations will be explained below.

Permission tracking and local permission inference. Our goal is to avoid annotations inside method bodies completely: based on the declared protocols, Plural infers how permissions flow through method bodies. Since Plural is based on a dataflow analysis, it automatically infers loop invariants as well.

However, Plural does require additional annotations on method parameters that have a declared protocol, such as the `ResultSet` parameter in figure 3. Notice that we use the same annotations for annotating parameters in client code that we use for declaring API protocols. While protocol annotations on the API itself (e.g., figure 2) can conceivably be provided by the API designer and amortize over the many uses of that API, the annotation shown in figure 3 is specific to this client program. In section 6 we discuss the overhead of providing these additional annotations for two open-source code bases.

Annotations make the analysis modular: Plural checks each method separately, temporarily trusting annotations on called methods and checking their bodies separately. For checking a given method or constructor, Plural assumes the permissions required by the method's annotations, i.e., it assumes the declared pre-condition. At each call site, Plural makes sure that permissions required for the call are available, splits them off (these permissions are "consumed" by the called method or constructor), and merges permissions ensured by the called method or constructor back into the current context. Notice that most methods "borrow" permissions (cf. figure 2), which means that they are both required and ensured. At method exit points, Plural checks that permissions ensured by its annotations are available, i.e., it checks the declared post-condition.

Thus, permissions are handled by Plural akin to conventional Java typing information: Permissions are provided with annotations on method parameters and then tracked automatically through the method body, like conventional types for method parameters. Unlike with Java types, local variables do not need to be annotated with permissions; instead, their permissions are inferred by Plural. Permission annotations can be seen as augmenting method signatures. They do not affect the conventional Java execution semantics; instead, they provide a static guarantee of protocol compliance without any runtime overhead.

Figure 3 shows a simple client method that retrieves an integer value from the first column in the first row of the given result set. Plural can be used to check whether this code respects the protocol declared for the `ResultSet` interface in figure 2. (It does not!)

API implementation checking. Our approach not only allows checking whether a client of an API follows the protocol required by that API, it can also check that the implementation of the protocol is consistent with its specification. The key abstraction for this is the *state invariant*, which we adapted from Fugue [17]. A state invariant associates a typestate of a class with a predicate over the fields of that class. In our approach, this predicate usually consists of access permissions for fields. An example can be found in figure 7, and details on the semantics of state invariants can be found in previous work [7].

Method cases. The idea of method cases goes back to behavioral specification methods, e.g., in the JML [27]. 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 [6, 4], but we are not aware of any other protocol checking

```
public static int getFirstInt(@Full(guarantee = "open") ResultSet rs)
{
    Integer result = null;
    if(rs.next()) {
        result = rs.getInt(1);
        if(rs.wasNull())
            result = null;
        return result;
    }
    else {
        return rs.getInt(1); // ERROR: rs in "end" instead of "valid"
    }
}
```

Fig. 3. Simple `ResultSet` client with error in *else* branch that is detected by Plural.

approaches that support method cases. In order to support method cases, Plural supports tracking disjunctions of possible permissions.

Branch sensitivity. APIs often include methods whose return value indicates the current state of an object, which we call *dynamic state tests*. For example, `next` in figure 2 is specified to return `true` if the cursor was advanced to a *valid* row and `false` otherwise.

In order to take such tests into account, 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*. For example, Plural updates the result set's state to *unread* at the beginning of the outer *if* branch in figure 3. Likewise, Plural updates the result set's state to *end* in the *else* branch and, consequently, signals an error on the call to `getInt`.

Notice that this approach does not make Plural path-sensitive: analysis information is still joined at control-flow merge points. Thus, at the end of figure 3, Plural no longer remembers that there was a path through the method on which the result set was *valid*. We believe that Plural could be extended to retain this information, but then we would have to deal with the usual complications of path sensitivity, i.e., large or infinite numbers of paths even through small methods.

When checking the implementation of a state test method, Plural checks at every method exit that, assuming `true` (or `false`) is returned, the receiver is in the state indicated by `true` (resp. `false`). This approach can be extended to other return types, although reasoning about predicates such as integer ranges may require using a theorem prover [8].

Dependent objects. Another feature of many APIs is that objects can become invalid if other, related objects are manipulated in certain ways. For example, SQL query results become invalid when the originating database connection is closed. (A similar problem, called concurrent modification, exists with iterators

[4].) There are no automated modular protocol checkers that we know of that can handle these protocols, although recent global protocol checking approaches can [10, 30].

Our solution is to "capture" a permission in the dependent object (the result set in the example) which prevents the problematic operation (closing the connection in the example) from happening. The dependent object has to be invalidated before "releasing" the captured permission and re-enabling the previously forbidden operation.

Captured permissions are typically required by a method but not returned. We use `@Perm` annotations for these situations, which allow declaring permissions required and ensured by a method separately (unlike `@Full` and similar annotations, which borrow a permission). Additionally, `@Capture` tells Plural to keep tracking the captured permission as a dependent of the capturing object, i.e., the method or constructor result. That allows explicitly releasing captured permissions with `@Release`. Additionally, such permissions will be implicitly released when the capturing object is no longer used. Released permissions become available to the client program again. For instance, `executeQuery` in figure 5 captures a receiver permission in the returned result set instance, which is explicitly released with `close` (figure 2) or implicitly when the result set is no longer used.

Others have modeled dependent objects with linear implications [12, 25, 21] but it is unclear how well those approaches can be automated. Our solution is to use a live variable analysis to detect *dead objects*, i.e., dead references to objects with `unique` permissions, and implicitly release any captured permissions from these dead objects.[3]

3 JDBC: Specifying a Java API

The Java Database Connectivity (JDBC) API defines a set of interfaces that Java programs can use to access relational databases with SQL commands. Database vendors provide *drivers* for their databases that are essentially implementations of the JDBC interfaces. Database client applications access databases primarily through `Connection`, `Statement`, and `ResultSet` objects. Clients first acquire a `Connection` which typically requires credentials such as a username and password. Then clients can create an arbitrary number of `Statement`s on a given connection. Statements are used to send SQL commands through the connection. Query results are returned as `ResultSet` objects to the client. Conventionally, only one result set can be open for a given statement; sending another SQL command "implicitly closes" or invalidates any existing result sets for that statement.

This section discusses the specification of these major interfaces (including subtypes) using Plural annotations. The specified interfaces are massive: they define over 400 methods, each of which is associated with about 20 lines of

[3] We could delete these objects (in C or C++) or mark them as available for garbage collection (in Java or C#), but we are not exploring this optimization possibility here.

JDBC interface	Lines (Increase)	Methods	On methods	State space	Total	Mult. cases
Connection	1259 (9.8%)	47	84	4	88	2
Statement	936 (9.4%)	40	64	2	66	0
PreparedStatement	1193 (5.5%)	55	58	0	58	0
CallableStatement	2421 (5.0%)	111	134	1	135	0
ResultSet	4057 (15.4%)	187	483	8	491	82
Total	9866 (10.4%)	440	823	15	838	84

Table 2. Specified JDBC interfaces with total lines, size increase due to annotations, methods, annotation counts (on methods, for defining state spaces, and total), and the use of multiple method cases in each file. Note that each file's length is almost entirely due to extensive informal documentation.

```
@States({"open", "closed"})
public interface Connection {

@Capture(param = "conn")
@Perm(requires = "share(this, open)", ensures = "unique(result) in open")
Statement createStatement() throws SQLException;

@Full(ensures = "closed")
void close() throws SQLException;

@Pure
@TrueIndicates("closed")
boolean isClosed() throws SQLException; }
```

Fig. 4. Simplified JDBC `Connection` interface specification.

informal documentation in the source files themselves, for a total of almost 10,000 lines including documentation (see table 2).

Connections. The `Connection` interface primarily consists of methods to create statements, to control transactional boundaries, and a `close` method to disconnect from the database (figure 4). Closing a connection invalidates all statements created with it, which will lead to runtime errors when using an invalidated statement. Due to space limits, we do not discuss our specification of transaction-related features here, but they are included in table 2.

Our goal was to specify JDBC in such a way that statements and result sets are invalidated when their connections are closed. Our solution is a variant on our previous work with iterators [4, 7]: we capture a `share` connection permission each time a statement is created on it. The captured permission has the *open* state guarantee, which guarantees that the connection cannot be closed while the statement is active. Plural releases the captured connection permission from a statement that is no longer used or when the statement is closed, as explained

```
@Refine({
  @States({"open", "closed"}),
  @States(refined="open", value={"hasResultSet","noResultSet"}, dim="rs") })
@Param(name = "conn", releasedFrom = "open")
public interface Statement {

@Capture(param = "stmt")
@Perm(requires = "full(this, open)", ensures = "unique(result) in scrolling")
ResultSet executeQuery(String sql) throws SQLException;

@Share("open")
int executeUpdate(String sql) throws SQLException;

@Full("open")
@TrueIndicates("hasResultSet")
@FalseIndicates("noResultSet")
boolean execute(String sql) throws SQLException;

@Capture(param = "stmt")
@Perm(requires = "full(this, open) in hasResultSet",
        ensures = "unique(result) in scrolling")
ResultSet getResultSet() throws SQLException;

@Full(value = "open")
@TrueIndicates("hasResultSet")
@FalseIndicates("noResultSet")
boolean getMoreResults() throws SQLException;

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

Fig. 5. JDBC `Statement` interface specification (fragment).

in section 2.2. When all statements are closed then a full permission for the connection can be re-established, allowing `close` to be called.

Statements. Statements are used to execute SQL commands. Statements define methods for running queries, updates, and arbitrary SQL commands (figure 5).

We specify `executeQuery` similarly to how statements are created on connections. The resulting `ResultSet` object captures a full permission to the statement, which enforces the requirement that only one result set per statement exists. Conversely, `executeUpdate` borrows a share statement permission and returns the number of updated rows. Since share and full permissions cannot exist at the same time, result sets have to be closed before calling `executeUpdate`. The `Statement` documentation implies that result sets should be closed before an update command is run, and our specification makes this point precise.

The method `execute` can run any SQL command. If it returns `true` then the executed command was a query, which we indicate with the state *hasResultSet*. `getResultSet` requires this state and returns the actual query result.

In rare cases a command can have multiple results, and `getMoreResults` advances to the next result. Again, `true` indicates the presence of a result set. We use a full permission because, like `execute` methods, `getMoreResults` closes any active result sets, as stated in that method's documentation: "Moves to this `Statement` object's next result, returns `true` if it is a `ResultSet` object, and implicitly closes any current `ResultSet` object(s) obtained with the method `getResultSet`."

Besides a plain `Statement` interface for sending SQL strings to the database, JDBC defines two other flavors of statements, prepared and callable statements. The former correspond to pattern into which parameters can be inserted, such as search strings. The latter correspond to stored procedures.

Since these interfaces are subtypes of `Statement` they inherit the states defined for `Statement`. The additional methods for prepared statements are straightforward to define with these states, while callable statements need an additional state distinction for detecting NULL cell values.

Overall, we were surprised at how well our approach can capture the design of the `Statement` interface.

Result sets. `ResultSet` is the most complex interface we encountered. We already discussed its most commonly used features in section 2. In addition, result sets allow for random access of their rows, a feature that is known as "scrolling". Scrolling caused us to add a *begin* state in addition to the *valid* and *end* states. Furthermore, the cell values of the current row can be updated, which caused us to add orthogonal substates inside *valid* to keep track of *pending* updates (in parallel to *read* and *unread*, see figure 2).

Finally, result sets have a buffer, the "insert row", for constructing a new row. The problem is that, quoting from the `ResultSet` documentation for `moveToInsertRow`, "[o]nly the updater, getter, and `insertRow` methods may be called when the cursor is on the insert row." Thus, scrolling methods are not available while on the insert row, *although the documentation for these methods does not hint at this problem.*

Our interpretation is to give result sets two modes (i.e., states), *scrolling* and *inserting*, where the former contains the states for scrolling (shaded in figure 1) as substates. `moveToInsertRow` and `moveToCurrentRow` switch between these modes. In order to make the methods for updating cells applicable in both modes we use method cases which account for all 82 methods with multiple cases in `ResultSet` (see table 2).

Figure 6 shows a fragment of the `ResultSet` interface with our actual protocol annotations. Notice how the two modes affect the methods previously shown in

figure 2. The figure also shows selected methods for scrolling, updating (including method cases), and inserting.[4]

4 Beehive: Verifying an intermediary library

This section summarizes a case study in using Plural for checking API compliance in a third-party open source code base, Apache Beehive. In the process we specified protocols for several other APIs besides JDBC (see section 3) including a simple protocol for Beehive itself.

Beehive[5] is an open-source library for declarative resource access. We have focused on the part of Beehive that accesses relational databases using JDBC. Beehive clients define Java interfaces and, using Java annotations, choose which SQL commands should be executed when a method in those interfaces is called. Notice that this design is highly generic: the client-specified SQL commands can include parameters that are filled with the parameters passed to the associated method. Beehive then generates code implementing the client-defined interfaces that simply calls a generically written SQL execution engine, `JdbcControl`, whose implementation we discuss below.

We first describe the APIs used by Beehive before discussing the challenges in checking that Beehive correctly implements a standard Java and its own API.

4.1 Checked Java standard APIs

We specified four Java standard APIs used by Beehive, highlighting Plural's ability to treat APIs orthogonally.

JDBC. We described the JDBC specification in section 3. Since Beehive has no apriori knowledge of the SQL commands being executed (they are provided by a client), it uses the facilities for running "any" SQL command described in section 3. Its use of result sets is limited to reading cell values, and a new statement is created for every command. We speculate that the Beehive developers chose this strategy in order to ensure that result sets are never rendered invalid from executing another SQL command, which ends up helping our analysis confirm just that.

Beehive is tricky to reason about because it aliases result sets through fields of various objects. Plural's modular approach nonetheless allowed us to move outwards from methods calling into JDBC to callers of those methods. In other words, we followed a process of running Plural "out of the box" on a given Beehive class first. Places where Plural issued warnings usually required annotations on method parameters (or for state invariants). Running Plural again would possibly result in warnings on the methods calling the previously annotated methods. Providing annotations for these methods would move the warnings

[4] `updateInt` defines two cases, which are both based on a borrowed full permission. One case requires that permission in the *valid* state and ensures *pending*, while the other case requires and ensures *insert*.

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

```
@Refine({
  @States({"open", "closed"}),
  @States(refined = "open", value = {"scrolling", "inserting"}),
  @States(refined = "scrolling", value = {"begin", "valid", "end"}, dim = "row"),
  @States(refined = "valid", value = {"read", "unread"}, dim = "access"),
  @States(refined = "valid", value = {"noUpdates", "pending"}, dim = "update")
  /*...*/ })
@Param(name = "stmt", releasedFrom = "open")
public interface ResultSet {

// changes from figure 2

@Full(guarantee = "scrolling", requires = "noUpdates")
@TrueIndicates("valid")
boolean next() throws SQLException;

@Full(guarantee = "scrolling", requires = "valid", ensures = "read")
int getInt(int columnIndex) throws SQLException;

@Pure(guarantee = "scrolling", requires = "read", ensures = "read")
boolean wasNull() throws SQLException;

// scrolling and updating result sets

@Pure("open")
@TrueIndicates("begin")
boolean isBeforeFirst() throws SQLException;

@Full(guarantee = "open", requires = "updatable")
@Cases({
  @Perm(requires = "this in valid", ensures = "this in pending"),
  @Perm(requires = "this in insert", ensures = "this in insert")
})
void updateInt(int columnIndex, int x) throws SQLException;

@Full(guarantee = "scrolling", requires = "pending", ensures = "noUpdates")
void updateRow() throws SQLException;

@Full(guarantee = "open", ensures = "insert")
void moveToInsertRow() throws SQLException; }
```

Fig. 6. JDBC `ResultSet` interface specification (fragment).

again until a calling method was able to provide the required permissions by itself because it created the needed API object.

Collections API. Beehive generically represents a query result row as a map from column names to values. One such map is created for each row in a result set and added to a list which is finally returned to the client.

The Java Collections API defines common containers such as lists and maps. *Iterators* are available for retrieving all elements in a container one by one. Maps provide *views* of their keys, values, and key-value pairs as sets. Lists support sub-list views that contain a subsequence of the list's elements. Views are "backed" by the underlying container, i.e., changes to the view affect the underlying container.

We specified the Collections API following our previous work [7]. Iterators are challenging because they do not tolerate "concurrent modification" of the underlying collection. We address this problem by capturing a immutable collection permission in the iterator [4, 7]. Views can be similarly handled by capturing a permission from the underlying collection when creating the view.

Regular expressions. Regular expressions are only used once in Beehive. The pattern being matched is a static field in one of Beehive's classes, which we annotate with `@Imm`.

The API includes two classes. A `Pattern` is created based on a given regular expression string. Then, clients call `find` or `match` to match the pattern in a given string. The `Matcher` resulting from these operations can be used to retrieve details about the current match and to find the next matching substring.

We easily specified this protocol in Plural. As with iterators, we capture a immutable `Pattern` permission in each `Matcher`. We use a typestate *matched* to express a successful match and require it in methods that provide details about the last match.

Exceptions. When creating an exception, a "cause" (another exception) can be set *once*, either using an appropriate constructor or, to our surprise, using the method `initCause`. The latter is useful when using exceptions defined before causes were introduced in Java 1.4. Beehive uses `initCause` to initialize a cause for such a legacy exception, `NoSuchElementException`. This protocol is trivial to specify in Plural, but it was fascinating that even something as simple as exceptions has a protocol.

Recurring patterns. There were at least three common challenges that we found across several of the APIs we specified.

1. We were surprised how prevalent *dynamic state test* methods are, and how important they are in practice. We found dynamic state test methods in JDBC, Collections, and regular expressions, and a large number of them in JDBC alone. For example, the method `hasNext` in the Java `Iterator` interface tests whether another element is *available* ([4], cf. section 4.2), and `isEmpty` tests whether a collection is *empty*. It was crucial for handling the Beehive code that our approach can express and benefit from the tests that are part of JDBC's facilities for executing arbitrary SQL commands.

2. We also found protocols involving multiple *interdependent objects* in these APIs (and very prevalent in JDBC). We could model these protocols by capturing and later releasing permissions.
3. We used *method cases* in JDBC and the Collections API. As previously shown, method cases can be used to specify full Java iterators, which may modify the underlying collection [4].

We believe that these are crucial to address in any practical protocol checking approach; our approach was expressive enough to handle these challenges for all the examples in our case study (see section 3).

4.2 Protocol Implementations

This section summarizes challenges in checking that Beehive implements the Java iterator API and that Beehive's main class is implemented correctly assuming clients follow Beehive's API protocol.

Implementing an iterator. Beehive implements an `Iterator` over the rows of a result set. Figure 7 shows most of the relevant code. We use state invariants, i.e., predicates over the underlying result set (see section 2.2), to specify iterator states. Notice that *alive* is our default state that all objects are always in. Thus its state invariant is a conventional class invariant [27, 2] that is established in the constructor and preserved afterwards.

When checking the code as shown, Plural issues 3 warnings in `hasNext` (see table 3). This is because our vanilla iterator specification [7] assumes `hasNext`, which tests if an element can be retrieved, to be pure. Beehive's `hasNext`, however, is not pure because it calls `next` on the `ResultSet`!

The way to fix this problem depends on whether or not you believe the `hasNext` method is supposed to be pure in all cases. If you believe it should be pure, you could modify Beehive's implementation of the iterator interface so that all effects are performed in the `next` method. Alternatively, you can specify the `hasNext` method as requiring a full permission, which we have done, and which causes the warnings to disappear.

Note that `next`'s specification requires *available*, which guarantees that `_primed` is `true` (see figure 7), making the initial check in `next` superfluous (if all iterator clients were checked with Plural as well).

Formalizing Beehive client obligations. Beehive is an intermediary library for handling resource access in applications: it uses various APIs to access these resources and defines its own API through which applications can take advantage of Beehive. We believe that this is a very common situation in modern software engineering: application code is arranged in layers, and Beehive represents one such layer. The resource APIs, such as JDBC, reside in the layer below, while the application-specific code resides in the layer above, making applications using Beehive appear like an hourglass.

```
@ClassStates({
  @State(name="alive",
    inv="full(_rs,scrolling) && full(_rowMapper) in init &&
         _primed == true => _rs in valid"),
  @State(name="available", inv="_primed == true") })
@NonReentrant
public class ResultSetIterator implements java.util.Iterator {
  private final ResultSet _rs;
  private final RowMapper _rowMapper;
  private boolean _primed = false;

  @Pure(guarantee = "next", fieldAccess = true)
  @TrueIndicates("available")
  public boolean hasNext() {
    if (_primed) { return true; }

    try {
        _primed = _rs.next();
    } catch (SQLException sqle) { return false; }
    return _primed;
  }

  @Full(requires = "available", ensures = "hasCurrent", fieldAccess = true)
  public Object next() {
    try {
      if (!_primed) {
        _primed = _rs.next();
        if (!_primed) {
          throw new NoSuchElementException();
        }
      }
      // reset upon consumption
      _primed = false;
      return _rowMapper.mapRowToReturnType(/* analysis-only */ _rs);
    } catch (SQLException e) {
        // Since Iterator interface is locked, all we can do
        // is put the real exception inside an expected one.
        NoSuchElementException xNoSuch = new NoSuchElementException("
            ResultSet exception: " + e);
        xNoSuch.initCause(e);
        throw xNoSuch;
    }
} }
```

Fig. 7. Beehive's iterator over the rows of a result set (constructor omitted). Plural issues warnings because `hasNext` is impure.

Beehive's API is defined in the `JdbcControl` interface, which `JdbcControlImpl` implements. `JdbcControlImpl` in turn is a client to the JDBC API. `JdbcControlImpl` provides three methods `onAcquire`, `invoke`, and `onRelease` to clients. The first one creates a database connection, which the third one closes. `invoke` executes an SQL command and, in the case of a query, maps the result set into one of several possible representations. One representation is the iterator mentioned above; another one is a conventional `List`. Each row in the result is individually mapped into a map of key-value pairs (one entry for each cell in the row) or a Java object whose fields are populated with values from cells with matching names.

Notice that some of these representations, notably the iterator representation, of a result require the underlying result set to remain open. The challenge now is to ensure that `onRelease` is not called while these are still in use because closing the connection would invalidate the results. This requirement is identical to the one we described for immediate clients of `Connection`, and thus we should be able to specify it in the same way.

However, the connection is in this case a field of a surrounding Beehive `JdbcControlImpl` object, and Plural has currently no facility for letting `JdbcControlImpl` clients keep track of the permission for one of its fields. Therefore, we currently work with a simplified `JdbcControlImpl` that always closes result sets at the end of `invoke`. Its specification, as desired, enforces that `onAcquire` is called before `onRelease` and `invoke` is only called "in between" the other two. This, however, means that our simplified `JdbcControlImpl` does not support returning iterators over result sets to clients, since they would keep result sets open. Overcoming this problem is discussed in the next section.

As mentioned, Beehive generates code that calls `invoke`. The generated code would presumably have to impose usage rules similar to the ones for `invoke` on *its* clients. Plural could then be used to verify that the generated code follows `JdbcControlImpl`'s protocol.

5 PMD: Scalability

We used the version of PMD included in the DaCapo 2006-10-MR2 benchmarks[6] to investigate how Plural can be used to check existing large code bases. In the next section this case study is used for direct comparison with state-of-the-art global protocol analyses [10, 30], which typically focus on simple protocols such as the well-known iterator protocol. Iterators are widely used in PMD, and most iterations in PMD are over Java Collections (see section 4.1), but PMD implements a few iterator classes over its own data structures as well.

Iterator protocol. We decided to focus on the simple and well-known iterator protocol (see section 4.1). It took one of the authors 75 minutes to examine and specify PMD, a specification that ultimately consisted of just 15 annotations.

[6] `http://dacapobench.org/`

This then enabled Plural to check that this protocol is followed all across PMD, which includes 170 distinct calls to the `next` method defined in the `Iterator` interface. Most iterator usages could be verified by Plural without any additional annotations because they are entirely local to a method. Annotations were needed where iterators were returned from a method call inside PMD and then used elsewhere. In one place an iterator is passed to a helper method *after* checking `hasNext`, and we could express the contract of this helper method with a suitable annotation.

Iterator implementations. PMD implements three iterators of its own. In one of them, `TreeIterator`, the implementation of `hasNext` is not only impure, like Beehive's iterator, but advances the iterator every time it is called. Thus, failure to call `next` after `hasNext` results in lost elements. The other iterators exhibit behavior compatible with the conceptual purity of `hasNext`: `next` is used to pre-fetch the element to be returned the *next* time it is called before returning the current element. `hasNext` then simply checks the pre-fetched element is valid, which is typically a pure operation.

In light of these and the iterator implementation in Beehive (figure 7), it appears legitimate to ask whether `hasNext` is really a pure operation. This would have significant consequences for behavioral specification approaches like the JML [27] or Spec# [2] because they use pure methods in specifications. Conventionally, the specification of `next` in the JML would be "`requires hasNext()`", but that would be illegal if `hasNext` was not pure. In contrast, our specifications are more robust to the non-purity of `hasNext`. In fact, Plural can verify iterator usage in PMD with a full permission for `hasNext` with the same precision.

6 Evaluation

This section summarizes overhead and precision of applying Plural to Beehive and discusses improvements to the tool to address remaining challenges.

Annotation overhead: The price of modularity. The overhead for specifying *Beehive* is summarized in table 3. We used about 1 annotation per method and 5 per Beehive class, for a total of 66 annotations in more than 2,000 lines, or about one annotation every 30 lines. Running Plural on the 12 specified Beehive source files takes about 34 seconds on a 800 Mhz laptop with 1GB of heap space for Eclipse including Plural.

For *PMD* we mentioned in section 5 that we only needed 15 annotations in total, which one of the authors provided in approximately 75 minutes. Thus, checking the iterator protocol was straightforward and imposed almost no overhead. Running Plural on PMD's entire codebase of 40KLOC in 446 files (with the same configuration as for Beehive) takes about 15 minutes.

Precision: A benefit of modularity. Plural reports 9 problems in *Beehive*. Three of them are due to the impure `hasNext` method in `ResultSetIterator` (see section

Beehive class	Lines / Methods	Annotations Meths.	Invs.	Total	Plural warnings	False pos.
DefaultIteratorResultSetMapper	37 / 2	1	0	1	0	0
DefaultObjectResultSetMapper	127 / 2	2	0	2	0	0
JdbcControlImpl	521 / 13	13	1	14	2	1
ResultSetHashMap	85 / 9	9	0	9	0	0
ResultSetIterator	106 / 4	4	3	7	3	0
ResultSetMapper	32 / 2	2	0	2	0	0
RowMapper	260 / 5	9	1	10	0	0
RowMapperFactory	156 / 7	3	0	3	4	4
RowToHashMapMapper	57 / 2	4	1	5	0	0
RowToMapMapper	49 / 2	4	1	5	0	0
RowToObjectMapper	236 / 3	4	0	4	0	0
SqlStatement	511 / 14	4	0	4	0	0
Total	2158 / 65	59	7	66	9	5

Table 3. Beehive classes checked with Plural. The middle part of the table shows annotations (on methods, invariants, and total) added to the code. The last 2 columns indicate Plural warnings and false positives.

4.2). Letting `hasNext` use a full permission removes these warnings. Another warning in `JdbcControlImpl` is caused by an assertion on a field that arguably happens in the wrong method: `invoke` asserts that the database connection is open before delegating the actual query execution to another, "protected" method that uses the connection. Plural issues a warning because a subclass could override one, but not the other, of these two methods, and then the state invariants may no longer be consistent. The warning disappears when moving the assertion into the protected method. Furthermore we note our state invariants guarantee that the offending runtime assertion succeeds.

The remaining warnings issued by Plural are false positives. This means that our false positive rate is is around 1 per 400 lines of code. We consider this to be quite impressive for a behavioral verification tool applied to complicated APIs (JDBC and others) and a very challenging case study subject (Beehive).

The false positive rate in *PMD* is extremely low. Warnings remained only in three places where PMD checks that a set is non-empty before creating an iterator and immediately calling `next` to get its first element. This is also mentioned as a source of imprecision in the most recent global protocol compliance checkers, which check for the same iterator protocol in PMD with 6 [10] and 2 [30] remaining warnings, respectively.

Future improvements. The remaining warnings in Beehive fall into the following categories:

- *Reflection (1).* Plural currently cannot assign permissions to objects created using reflection in `RowMapperFactory`.

– *Static fields (3).* `RowMapperFactory` manipulates a static map object, which we specified to require full permissions. For soundness, we only allow duplicable permissions, i.e., share, pure, and immutable, on static fields.
– *Complex invariant (1).* `JdbcControlImpl` opens a new database connection in `onAcquire` only if one does not already exist. We currently cannot express the invariant that a non-`null` field implies a permission for that field, which would allow Plural to verify the code.

These are common sources of imprecision in static analyses. We are considering tracking fields as implicit parameters in method calls, as discussed in section 4.2, and static fields could be handled in this way as well. Related to this issue is also a place in Beehive where a result set that was assigned to a field in the constructor is implicitly passed in a subsequent method call. We turned it into an explicit method parameter for now (the call to `mapRowToReturnType` in figure 7). Java(X) has demonstrated that fields can be tracked individually [14], although we would like to track permissions for "abstract" fields that do not necessarily correspond to actual fields in the code. We are also working on a strategy for handling object construction through reflection, and on generalizing the state invariants expressible in Plural.

We also simplified the Beehive code in a few places where our approach for tracking local aliases leads to analysis imprecisions. Since local alias tracking is orthogonal to tracking permissions we used the simplest available, sound solution in Plural, which is insufficient in some cases. We plan to evaluate other options.

Problems occur when the same variable is assigned different values on different code paths, usually depending on a condition. When these code paths rejoin, Plural assumes that the variable could point to one of several locations, which forbids strong updates. We are investigating using more sophisticated approaches that avoid this problem. Alternatively, Plural will work fine when the part of the code that initializes a variable on different paths is refactored into a separate method. Notice, however, that tracking local aliasing is a lot more tractable than tracking aliasing globally. Permissions reduce the problem of tracking aliasing globally to a local problem.

Finally, we assumed one class to be non-reentrant, but we believe a more complicated specification would allow the class to be analyzed assuming reentrancy. Our approach conservatively assumes that all classes are re-entrant [7]—meaning that within the dynamic scope of a method, another method of the same class may be invoked with the same receiver object—but in practice that is not always the developer's intention. Therefore, we use the (currently unchecked) annotation shown in figure 7 to mark a class as non-reentrant, which causes Plural to omit certain checks during API implementation checking. We are planning on checking this annotation with Plural in the future.

Refactoring option. Notice that besides improving the tool there is usually the option of refactoring the problematic code. We believe that this is an indicator for the viability of our approach in practice, independent of the features supported by our tool: developers can often circumvent tool shortcomings with

(fairly local) code changes. On the other hand, we have not seen many examples that fundamentally could not be handled by our approach.

Iterative process. We noticed that Plural has several characteristics that seem to facilitate its retroactive use with existing code. First, running Plural on unannotated API client code will result in warnings on some of the calls into the API. Removing these warnings requires annotating the client methods in question, which will "move" the warnings to where these methods are invoked. This process continues until it reaches code where API objects are created. In the case of iterators, that is often the method where they are also used, in which case no developer intervention is required. Second, our experience also suggests that checking protocols for different APIs is largely orthogonal. Finally, annotations allow making assumptions about parts of the codebase that one wants to ignore for the time being, for instance because that part of the code is known not to interfere with the API protocol at hand.

7 Related Work

We previously proposed access permissions for sound, modular typestate protocol enforcement in the presence of aliasing, first for single-threaded [7] and recently for multi-threaded programs [3]. We showed on paper that the proposed type systems can handle interesting protocols, including iterators. We also developed Plural, an automated tool that embodies our permission-based approach as a static dataflow analysis for Java [9]. A comprehensive description of the Plural tool is part of the first author's dissertation [5]. This paper evaluates our approach for specifying and checking compliance to API protocols using Plural.

A plethora of approaches was proposed in the literature for checking protocol compliance and program behavior in general. These approaches differ significantly in the way protocols are specified, including typestates [33, 15, 26, 19, 17, 7], type qualifiers [20], size properties [13], direct constraints on ordering [24, 10, 30], type refinements [29, 14], first-order [27, 2] or separation logic [32], and various temporal logics [23]. In these approaches, like in ours, usage rules of the API(s) of interest have to be codified by a developer. Once usage protocols are codified, violations can be detected statically (like in our and most of the above approaches) or dynamically (while the program is executing, e.g. [6, 18]).

Many of the proposed static approaches, including ours, are modular and require developer-provided annotations in the analyzed code in addition to codifying API usage rules (e.g. [17, 14]) but there are also global approaches that require no or minimal developer intervention (e.g. [20, 23]). Unlike previous modular approaches, our approach does not require precise tracking of all object aliases (e.g. [15, 17]) or impose an ownership discipline on the heap (e.g. [2]) in order to be modular.

Ours is one of the few approaches that can reason about correctly *implementing* APIs independent from their clients. (Interestingly, all of these approaches that we are aware of are modular typestate analyses [17, 26, 7].) Ours is the only

approach (that we are aware of) that can verify correct usage *and implementation* of dynamic state test methods. Several other approaches can verify their correct usage (e.g., [29, 13]), but not their implementation.

Previous modular approaches are often proven sound and shown to work for well-known examples such as file access protocols. But automated checkers are rare, and case studies with real APIs and third-party code hard to find. Notable exceptions include Vault [15] and Fugue [17, 16], which are working automated checkers that were used to check compliance to Windows kernel and .NET standard library protocols, respectively (although Vault requires rewriting the code into its own C-like language).

This paper shows that our approach can be used in practical development tools for enforcing real API protocols. As far as we know, this paper is the first one that reports on challenges and recurring patterns in specifying typestate protocols of large, real APIs. We also report overhead (in terms of annotations) and precision (in terms of false positives) in checking open-source code bases with our tool.

We suspect that empirical results are sparse because APIs such as the ones discussed in this paper would be difficult to handle with existing modular approaches due to their limitations in reasoning about aliased objects. These limitations make it difficult to specify the object dependencies we found in the JDBC, Collections, and Regular Expressions APIs in the Java standard library. Fugue, for instance, was used for checking compliance with the .NET equivalent of JDBC, but the published specification does not seem to enforce that connections remain open while "commands" (the .NET equivalent of JDBC "statements") are in use [16]. Existing work on permissions recognized these challenges [11, 12] but only supports unique and immutable permissions directly and does not track behavioral properties (such as typestates) with permissions.

In contrast to modular checkers, many global analyses have been implemented and empirically evaluated. While model checkers [23] typically have severe limitations in scaling to larger programs, approaches based on abstract interpretations have been shown to scale quite well in practice. "Sound" (see below) approaches rely on a global aliasing analysis [1, 19, 10, 30] and become imprecise when alias information becomes imprecise.

This paper shows that our approach at least matches the most recently proposed global analyses that we are aware of in precision when verifying iterator usage in PMD [10, 30] with extremely low developer overhead. Another previous global typestate analysis has also been used—with varying precision—to check simple iterator protocols, but in a different corpus of client programs [19].

These global typestate-based analyses have been used to make sure that dynamic state test methods are *called*, but not that the test actually indicated the needed state [19, 10, 30]. For example, the protocols being checked require calling `hasNext` before calling `next` in iterators, but they do not check whether `hasNext` returned `true`, which with our approach is expressed and ensured easily. Tracematch-based analyses [10, 30] currently lack the expressiveness to capture these protocols more precisely, while approaches based on must-alias information

(e.g. [19]) should be able to, but do not in their published case studies, encode these protocols. This is arguably an omission in these approaches that, given the importance of dynamic state tests in practice, we believe should be addressed.

Note that our approach, unlike global analyses, can reason about API implementations separately from clients and handles dynamic state tests soundly, as discussed above. Reasoning about API implementations separately from clients is critical for libraries such as Beehive that may have many clients. Our approach also seems to match the precision of global analysis for checking a simple iterator protocol. Additional empirical comparisons with global analyses can be found elsewhere [5].

Lastly, work has been done on inferring API usage protocols [31] and flagging deviations from commonly followed rules using statistical methods [28]. These approaches are complimentary to ours as the inferred protocols could be specified and checked with our approach.

8 Conclusions

This paper evaluates access permissions for enforcing API protocols using our prototype tool, Plural. It reports on our experience in specifying JDBC and several other important Java standard APIs, identifying common challenges for any practical API protocol enforcement technique. The paper also summarizes case studies in checking third-party open source applications *after the fact*, i.e., by using Plural on the existing code base, injecting annotations, and performing small refactorings. In future work we plan to evaluate Plural *during* software development.

Intermediary libraries, such as the one we consider in this paper, represent a compelling use case for Plural. Because Plural is modular *and* can verify implementations of protocols it can be used to verify the library by itself, assuming the specification of underlying APIs and imposing rules on potential clients but *without* depending on the specifics of a sample client or a concrete implementation of the underlying APIs. Thus, the effort for verifying a library can amortize across the users of the library *and* the possible combinations of underlying API implementations (such as the drivers for various databases).

To our knowledge, this is the first comprehensive evaluation of a modular protocol checking approach in terms of its ability to specify large, real APIs. We also report annotation overhead and precision in checking open-source code bases with our tool. We find that our approach imposes moderate developer overhead in the form of annotations on classes and methods and produces few false positives. These results indicate that our approach can be used to specify and enforce API protocols in practice. From specifying APIs we notice several recurring patterns including the importance of dynamic state tests, method cases, and the dependency of API objects on each other. The extremely small overhead of enforcing a simple protocol (iterators) in a large code base (PMD) also suggests that our approach can be introduced gracefully into existing projects, with increasing effort for increasingly interesting protocols.

Acknowledgments. We thank Ciera Jaspan, Joshua Sunshine, and the anonymous reviewers for feedback on drafts of this paper. This work was supported in part by a University of Coimbra Joint Research Collaboration Initiative, DARPA grant #HR0011-0710019, and NSF grant CCF-0811592. Nels Beckman is supported by a National Science Foundation Graduate Research Fellowship (DGE-0234630).

References

1. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *8th SPIN Workshop*, pages 101–122, May 2001.
2. 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.
3. 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.
4. K. Bierhoff. Iterator specification with typestates. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006.
5. K. Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, School of Computer Science, Apr. 2009.
6. 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.
7. 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.
8. K. Bierhoff and J. Aldrich. Permissions to specify the composite design pattern. In *7th International Workshop on Specification and Verification of Component-Based Systems*, Nov. 2008.
9. K. Bierhoff and J. Aldrich. PLURAL: Checking protocol compliance under aliasing. In *Companion of the 30th International Conference on Software Engineering*, pages 971–972. ACM Press, New York, May 2008.
10. 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.
11. J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.
12. 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.
13. W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *International Conference on Software Engineering*, pages 186–195, May 2005.
14. M. Degen, P. Thiemann, and S. Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*, pages 550–574. Springer, Aug. 2007.
15. 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.

16. R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.
17. R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.
18. M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *International Conference on Software Engineering*, pages 220–229. IEEE Computer Society, May 2007.
19. S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ACM International Symposium on Software Testing and Analysis*, pages 133–144, July 2006.
20. 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.
21. C. Haack and C. Hurlin. Resource usage protocols for iterators. In *International Workshop on Aliasing, Confinement and Ownership*, July 2008.
22. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
23. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.
24. A. Igarashi and N. Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages*, pages 331–342, Jan. 2002.
25. N. Krishnaswami. Reasoning about iterators with separation logic. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 83–86. ACM Press, Nov. 2006.
26. V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, Dec. 2006.
27. 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.
28. B. Livshits and T. Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 296–305, Sept. 2005.
29. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ACM International Conference on Functional Programming*, pages 213–225, 2003.
30. N. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 347–366, Oct. 2008.
31. M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 77–96, Oct. 2005.
32. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *ACM Symposium on Principles of Programming Languages*, pages 75–86, Jan. 2008.
33. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157–171, 1986.
34. P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.