Plural: A Typestate Checker Based on Access Permissions

Kevin Bierhoff
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
kevin.bierhoff@cs.cmu.edu

Nels E. Beckman
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
nbeckman@cs.cmu.edu

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

In this paper we present Plural, a static typestate checker for Java programs.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Delphi theory

Keywords

ACM proceedings, LaTeX, text tagging

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, and is compatible with almost all features of the Java programming language. Plural integrates into Eclipse's Java IDE and can perform typestate checking on entire projects in seconds. In this paper we will describe many of the various 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. For these reasons, 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 defined (knowingly or otherwise) as part of the class declaration process. One classic example of an obect protocol is Java's `FileInputStream` class in the java.io package. After construction, a file input stream object is open and the `read` method can be called, but after the point at which the `close` method has been called on an object, a method which releases certain system resources, subsequent calls to `read` will result in an `IOException` being thrown.

Object protocols are typically not checked by the compiler but rather described in prose comments and documentation accompagnying 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 [23, 10, 17, 9, 19, 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). Others promote effeciency (e.g., borrowing) or lower the burden of specification (e.g., packing inference). And still others 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, and in this work we will go into much more depth about the features of the tool, and cover many features that were not mentioned at all in earlier work.

This paper will proceed in the following manner. Section 2 gives a brief recap of the access permissions method of typestate checking. Section 3 describes the basic specification language, 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 and then we conclude.

2. ACCESS PERMISSIONS

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

Figure 1: Access permission taxonomy

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 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. If other references alias r, then at any method call we must conservatively assume that those references are used to modify the state of the object. In order to to get a handle on this problem, access permissions are each associated with 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 availble 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 us determine statically when all of the pieces of permission have been accounted for.

Finally, access permissions also support state hierarchies and guaranteed states within those hierarchies. State hierachies allow a class to define substates of existing states. For example, a stream class might have Open and Closed states. That Open state may be further refined to have 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.

```
@Full(requires="HasNext",ensures="alive")
@ResultShare
Object next() { ... }

static boolean hasNext(@Pure Iterator i) {
  return i.hasNext();
}
```

Figure 2: Basic specifications in Plural

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 encoded those permissions inside Plural.

3. 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, specifically the Java Development Tools project. Furthermore, Plural is an extension of Crystal [**?**], an educational framework for developing dataflow analyses. Plural is open-source and is freely available for download.[1]

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 of references 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 is 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 make use of three of these basic annotations.

The `next` method is a method of the `Iterator` interface. Here is the meaning of its specification: 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

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

```
@Refine({
  @States({"Open","Closed"}),
  @States(refined="Open",
          value={"Ready","EOF"}) })
class Stream { ... }

void read(@Share("Open") Stream s) {
  while(s.hasMore()) { s.read(); }
}
```

Figure 3: Declaring a state hierarchy for our stream

specification is ommitted, as in `@ResultShare`, alive is automatically used.

The `hasNext` static method requires a pure permission to its sole parameter, which will allow it to call read-only methods on the parameter. Again, because the annotation has no further specification, defaults are used; the iterator must be in the alive state to call the method and will be returned in the alive state. The pure permission will be returned to the caller when the method returns.

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 the case of Read and EOF, the Open state is refined (in other words, Read and EOF are substates of Open). Open and Closed, however, are not specified as being substates of any other 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 the stream will not leave the Open state, in other words, that it will not be closed. This prevents the method from calling the close method. However, in order to obtain a guaranteed permission, all aliases must be prevented from leaving the Open state, and therefore the code inside the `read` body can rely on the object not being closed.

Runtime Representation. Permissions in the Plural implementation are represented by a fraction function, a map from states in the state hierarchy 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. (In fact, 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:

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

The annotations presented in this section are the standard annotations in Plural, and the ones used most commonly for writing basic specifications. In the next section we describe some more advanced specifications that allow for greater expressiveness.

4. ADVANCED SPECIFICATIONS

4.0.1 Borrowing

Over and over again we encountered methods that *borrow* permissions, meaning they return the same permissions that are passed into them.[2] Technically, the inference system from section 5 cannot express borrowing because it only quantifies over fractions inside a method pre- or post-condition, but not both. But borrowed permissions are easily supported in Plural because they are essentially permissions whose fractions are universally quantified across both the method pre- and post-condition.

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. Without support for borrowing, there would be an error in figure **??**. The reason is that the given full permission is strong enough to introduce the *valid* state guarantee for `getInt`, but not to drop it afterwards, which is necessary to return the permission with the *open* state guarantee as declared. With borrowing, we know that we can restore the previous state 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 that was previously in the lattice. In practice, this cuts the time it takes our regression suite to run roughly in half (see figure 4).

4.0.2 Method Cases

The idea of method cases goes back to behavioral specification methods, e.g., in the JML [20]. 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 terms of linear logic, method cases can be thought of as a choice between implications. For example, $(P_1 \multimap P_2) \;\&\; (P_3 \multimap P_4)$ encodes a method with two cases $P_1 \multimap P_2$ and $P_3 \multimap P_4$.

In order to support method cases, Plural tracks the possible permissions after a call to a method with cases using

[2]The other common case is that permissions are *consumed*, i.e. not returned to the caller of a method.

"choice" lattice elements. Checking the implementation of a method with multiple cases amounts to checking the method separately for each case.

4.0.3 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 [15] and type refinements [11]. For example, result sets can be marked as *updatable* or *readonly* (see section **??**), and they cannot switch from one to the other once created. 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].

4.0.4 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 [2].) There are no automated modular protocol checkers that we know of that can handle these protocols, although recent global protocol checking approaches can [6, 21].

Permission Parameters. 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 declared in Plural with a `@Param` annotation, and `@Release` explicitly releases permissions, as in `close` (figure **??**).

Garbage Collection. Others have modeled dependent objects with linear implications [8, 18, 16] 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 release any captured permissions from these dead objects.[3]

4.0.5 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.

[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.

Inheritance. **{{{ TODO: XXX }}}**

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 lattice element. The constructor post-condition is developer-declared and checked as usual, and calls to `super`-constructors can be handled like calls to inherited methods.

Permissions required for virtual method invocations from inside a constructor have to be declared as usual; these permissions are required when subclass constructors invoke the constructor. This means that constructors have to *admit to* any virtual method calls they perform on the newly constructed object, which appears desirable considering the amount of grief such calls can cause to subclass developers. For soundness, the implementation of such constructors is checked twice: once assuming the constructor initializes an object of its own type (where the injected `unique` permission can be used for virtual method calls and declared receiver pre-conditions can be ignored) and once assuming it was called using `super` from a subclass constructor (where the explicitly required permissions are used for virtual method calls).

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 (cf. chapter **??**): frame permissions required by a private method can be satisfied with frame permissions available at the call site. This simplifies specifying the methods calling 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. Plural associates arrays with a permission of their own and makes sure that a modifying permission (`unique`, `full`, or `share`) is available upon stores into the array. Plural does *not* currently allow tracking permissions for the array elements, which is subject of future work. This issue is related to putting permissions into containers such as lists and sets. **{{{ TODO: Nels: has this changed with new quantifier support? }}}**

5. SPECIFICATION INFERENCE

5.0.6 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 code implementing an API is safe when used with its declared protocol. The key abstraction for this are *state invariants*, which we adapted from Fugue [10]. 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.

Whenever a receiver field is used, Plural *unpacks* a permission to the surrounding object to gain access to its state invariants [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]. Plural sometimes has to "guess" what state to pack to by trying all possibilities, which our "choice" contexts allow us to do.

5.1 Local Must Alias Analysis

Plural uses a simple local alias analysis to identify local variables that are known to point to the same object. The local alias analysis tracks a set of "locations" for each local variable. These locations are effectively Plural's static approximation of the different objects that a local variable may point to at run time. The tuple lattice described above therefore associates permission sets and constraints with these locations.

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.

The purpose of Plural's local alias analysis is therefore *not* to soundly approximate all possible runtime aliasing, but to identify local variables that must alias and can therefore access the same permissions. The local alias analysis in particular prevents us from having to split permissions when a local is assigned to another local with a *copy* instruction, such as `x = y`, which avoids developer-provided annotations in method bodies.

6. CONCRETE PREDICATES

Concrete predicates track information about *values* such as Booleans as well as the null-ness of references. Tracking the former is necessary for properly handling dynamic state tests, while the latter is useful in implementation predicates as well as for avoiding null dereference errors.

Every *test* in a program, for example in an *if* statement or loop header, ascertains the truth or falsehood of some (temporary) variable at run time. This information often *implies* or *indicates* some other fact. For example, truth of the `ResultSet`'s `next` method's return value indicates that the method call's receiver is in the `valid` state (see figure **??**).

Plural maintains a list of known implications from Boolean to other facts and eagerly eliminates them when knowledge about the value of a Boolean variable becomes available. Plural handles the fact that a reference is "null" or "non-null" similar to a Boolean fact about a variable. If null-ness is tested at run time, Plural produces an implication from a Boolean to a null-ness fact as described above.

Future Work: Integers. Facts about integer variables, such as an integer being positive, could conceivably be tracked by Plural as well. Unlike with Boolean and null-ness facts, however, integers are notoriously difficult to reason about, and facts about them hard to track precisely and efficiently. In the spirit of Plural, one could use sufficiently precise and efficient lattices to track integer ranges, as was recently shown feasible [13]. Alternatively, one could employ a theorem prover, as is common in behavioral verification [14], which would increase reasoning power but also overhead and might decrease predictability.

Discussion. Typestates are a promising framework for tracking concrete predicates more precisely. What we mean is that the tracking of concrete predicates discussed above is a well-studied area, often even commonly tracked with dataflow analyses based on simple lattices. But we found that concrete predicates often depend on an object's typestate: for instance, a field may only be non-null in a certain state and otherwise be null. We therefore believe that wrapping a typestate framework, such as Plural's, around these well-known techniques for tracking concrete predicates would allow tracking them in the appropriate context (e.g., assuming a particular typestate) and yield higher precision. In particular, considerable research effort has gone into reasoning about object initialization [12, 22], which we believe can be largely encoded with typestates.

At the same time, concrete predicates are crucial for reasoning about dynamic state tests, thereby helping our typestate analysis. While Plural only supports state test methods of Boolean return type, we have seen integers and null-ness as well as regular or exceptional control flow indicate states.

6.0.1 Dynamic State Tests

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 **??** 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 **??**. Likewise, Plural updates the result set's state to *end* in the *else* branch and, consequently, signals an error on the call to `getInt`. Recent whole-program protocol analyses [21, 6] would miss this error because they merely ensure that dynamic state tests are *called.*

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 **??**, Plural no longer remembers that there was a path through the method on which the result set was *valid*.

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 undecidable theories such as integer ranges may require using a theorem prover.

7. ERROR REPORTING

7.0.2 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 method call site whether the pre-condition is satisfiable. The actual error may precede the point in the method where a protocol violation occurs, for instance because an erroneous method call may set up a situation where a later pre-condition becomes unsatisfiable. But the error will always be in the method being identified by Plural, including a wrong declared pre-condition.

TODO Nels, can we summarize your work on error diagnosis tooling here?

8. EXPERIENCE

We have used Plural, the typestate verification tool described in Section **??**, 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 4 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 guage 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.[4] Figure 4 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[5] 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 inferface 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[6] 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.

Beehive. Apache Beehive[7] 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.

9. ACKNOWLEDGEMENTS

We would like to thank Paul Richardson for his work on the Fiddle protocol visualization.

References

[4]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.

[5]`http://pmd.sourceforge.net/`

[6]`http://code.google.com/p/fourinaline/`

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

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

[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. Iterator specification with typestates. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006.

[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] 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. 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] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *SIGPLAN Not.*, 36(5):59–69, 2001.

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

[11] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *ACM Symposium on Principles of Programming Languages*, pages 281–292, 2004. URL `citeseer.ist.psu.edu/dunfield04tridirectional.html`.

[12] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 337–350, Oct. 2007. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297052.

[13] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .NET. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 329–346, 2008.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, May 2002.

[15] 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. URL `citeseer.ist.psu.edu/foster02flowsensitive.html`.

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

[17] P. Joshi and K. Sen. Predictive typestate checking of multithreaded java programs. *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 288–296, Sept. 2008.

[18] 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.

[19] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.*, 39(3):46–55, 2004. ISSN 0362-1340.

[20] 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.

[21] 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.

[22] X. Qi and A. C. Myers. Masked types for sound object initialization. In *ACM Symposium on Principles of Programming Languages*, Jan. 2009. To appear.

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