Checking Concurrent Typestate with Access Permissions in Plural: A Retrospective

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich

Abstract Objects often define usage protocols that clients must follow in order for these objects to work properly. In the presence of aliasing, however, it is difficult to check whether all the aliases of an object properly coordinate to enforce the protocol. Plural is a type-based system that can soundly enforce challenging protocols even in concurrent programs. In this paper, we discuss how Plural supports natural idioms for reasoning about programs, leveraging *access permissions* that express the programmer's design intent within the code. We trace the predecessors of the design intent idioms used in Plural, discuss how we have found different forms of design intent to be complimentary, and outline remaining challenges and directions for future work in the area.

1 Introduction

Many libraries and components define *usage protocols*: constraints on the order in which clients may invoke their operations. For example, in Java one must first call `connect` on a `Socket`, after which data may be read or written to the socket. Once `close` is called, reading and writing is no longer permitted.

A recent corpus study of protocols in Java libraries showed that protocol definition is relatively common (in about 7% of types) and protocol use even more so (about 13% of classes) [2]. By comparison, only 2.5% of the types in the Java library define type parameters; thus, the commonality of protocol definition compares well

Kevin Bierhoff
Two Sigma Investments, e-mail: kevin.bierhoff@cs.cmu.edu

Nels E. Beckman
Google Pittsburgh, e-mail: nbeckman@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University, e-mail: jonathan.aldrich@cs.cmu.edu

to the use of an important Java language feature. Protocols also cause problems for developers in practice: Jaspan found that almost 20% of the understandable postings in an ASP.NET help forum were related to protocol constraints [10].

Over the past 6 years, we have been developing Plural, a type-based system for specifying and enforcing correct protocol usage. The primary goal of Plural is to make modular protocol checking practical for realistic object-oriented programs. Preliminary evidence suggests that we have made substantial progress towards this goal: Plural has been applied to multiple open source programs totaling 100kloc+, assuring hundreds of protocol uses and finding many protocol errors and race conditions in well-tested code [4, 2].

In this paper, we reflect on the key design characteristics that have enabled Plural to effectively verify realistic software. Many real program designs use substantial aliasing, which provides many design benefits, but which also makes reasoning about protocols difficult, as the multiple clients of an abstraction must coordinate to obey the protocol. Central to the success of our approach was drawing inspiration from how software developers naturally reason about protocols in the presence of aliasing. In order to provide scalability and modularity, our approach also provides natural ways for developers to express their design intent when specifying and verifying protocol usage.

In the next section, we will look at the historical development of protocol checking ideas, with a particular focus on design intent and developer reasoning. Section 3 describes how we further developed these ideas, reviewing the design of the Plural tool from earlier work. Section 4 reflects on our experience with Plural, both in terms of our successes in verifying usage protocols in challenging contexts, and in terms of the limitations we have observed in the methodology. We close in section 5 with a discussion of future research directions that may enable protocol checking to become a routine part of software development practice.

2 Historical Context

The idea that checking sequences of events is important in software reliability has a long history. Fosdick and Osterweil [13] were the first to suggest applying program analysis techniques to address the challenges of software reliability. Their Dave system used data flow analysis to detect "data flow anomalies" that are symptomatic of programming errors. These anomalies, such as dead stores to variables or uses of undefined variables, were expressed as problematic sequences of access to a variable: e.g. a read after creation, or two writes with no intervening read. Of course, an anomaly is not necessarily an error; Fosdick and Osterweil state that "a knowledge of the intent of the programmer is necessary to identify the error." Given the context, Fortran programming in the 1970s, it was "unreasonable to assume that the programmer will provide" that design intent. Still, the remark foreshadowed subsequent work that focused on how to make descriptions of limited design intent practical.

A major step in the direction of providing design intent regarding protocols came a decade later, when Strom and Yemini [26] proposed a *typestate* abstraction for enforcing protocol constraints. Their programming language, NIL, allowed the gradual initialization of data structures to be tracked statically. In order to make analysis modular, they proposed that programmers declare a simple form of design intent, specifying the initialization state of these data structures at procedure boundaries.

Olender and Osterweil [24] observed that it is insufficient to only track fixed properties like data initialization, saying "a flexible sequence analysis tool was needed." Their Cecil specification system allowed the analyst to define which events he or she wishes to reason about, to correlate those events to program statements, and then to specify a regular expression that describes a valid sequence of events. The Cesar analysis tool could then verify that the program obeyed the Cecil specification.

Because of the frequent aliasing in object-oriented programs, an application of typestate or sequence analysis in that context required significant advances in specifying and reasoning about aliases. The Turing language included an annotation (now commonly called unique) specifying that a particular argument to a function may not be aliased [16], meaning that program analysis tools need not worry about interference when tracking the state of an object. Hogg proposed the idea of *borrowed* arguments, which may not be stored in fields, and thus help to maintain uniqueness. In addition to borrowing, Hogg proposed read-only method arguments (which we call pure), which allow tools to assume that a method does not change the state of the passed-in object [15]. Noble et al. proposed immutable as a stronger version of pure, specifying that the referenced object cannot be changed through any reference in the program [23]. Following others, we use share to indicate an unrestricted reference, which may be aliased arbitrarily.

These three ideas—a programmer-defined sequence of events, an expression of sequencing design intent via typestate, and an expression of aliasing intent via alias annotations—first came together in DeLine and Fähndrich's Vault [11] project for verifying low-level software. Their later Fugue system [12] adapted protocol checking to the object-oriented setting, developing a methodology for modular typestate checking in the presence of inheritance. Despite these advances, however, Fugue was limited to reasoning about the state of unique objects; it was not able to help developers in the relatively common case of aliased objects with typestate. Our work was motivated by the need to overcome this limitation.

3 Typestate Protocols with Access Permissions

The goal of Plural is to modularly check protocols in realistic object-oriented programs. Checking realistic programs means that we need some way of reasoning about the aliasing that occurs in these designs. Modular checking means we would like to check code one method at a time, using only the specifications of other objects; this requires us to describe the protocol and aliasing state of objects at method

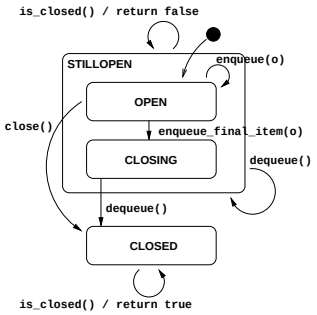

Fig. 1 Simplified `Blocking_queue` protocol. Rounded rectangles denote states refining another state. Arches represent method calls, optionally with return values.

boundaries. We accomplish both tasks using access permissions [6], which we have implemented in Plural [7], an automated tool for checking permission-based typestate protocols in Java. This section is a review of previously published work [6, 3, 7].

3.1 Access Permissions

Our static, modular approach to checking API protocols is based on access permissions ("permissions" for short). Permissions are predicates associated with program references describing the abstract state of that reference and the ways in which it may be aliased.

In our approach, developers start by specifying their protocol. Figure 1 shows a simplified protocol for a concurrent blocking queue `Blocking_queue`.[1] Its protocol is modeled as a Statechart [14]. Blocking queues can be used to hand work items from one producer thread to multiple consumer threads. We will use this protocol as a running example.

We allow developers to associate objects with a *hierarchy* of typestates, similar to Statecharts [14]. For example, *OPEN* and *CLOSING* (no more items can be enqueued) can be summarized into *STILLOPEN*, which is the state needed for consumer threads to `dequeue` an item (figure 1). Hierarchy serves to naturally encode design intent regarding the structure of the state space; we have found it to be essential for expressing complex protocols compactly [5].

Methods correspond to state transitions and are specified with *access permissions* that describe not only the state required and ensured by a method but also how the method will access the references passed into the method. We distinguish exclusive (unique), exclusive modifying (full), read-only (pure), immutable, and shared access (table 1). Furthermore, permissions include a *state guarantee*, a state that the method promises not to leave [6].

[1] We greatly appreciate Allen Holub for use of this example.

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

Permissions are associated with object references and govern how objects can be accessed through a given reference [6]. They can be seen as rely-guarantee contracts [18] between the current reference and all other references to the same object: permissions provide guarantees about object accesses through other references *and* restrict the current reference to not violate other permissions' assumptions. Thus our permissions encode a natural form of reasoning that has been used in concurrent systems, but adapt it to reason in the context of aliasing in addition to concurrency. Permissions capture three kinds of design intent:

1. *What kinds of references exist?* Our system distinguishes read-only and modifying access, both through the current reference and (if applicable) other references. The resulting 5 permissions (shown in Table 1) cover the space of possibilities in a more systematic way than previously described permission systems, and enable natural reasoning based on who is in control of a protocol.
2. *What state is guaranteed?* A guaranteed state supports natural reasoning about agreements ensuring that the multiple clients of an object do not interfere with each other. A client can rely on this guaranteed state even if the referenced object may be modified by other clients.
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) with many readers (pure); and only readers (immutable and pure) with no writers.

Permissions are linear in order to preserve this invariant. But unlike linear type systems [27], 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* [9] we can also *merge* previously split permissions when aliases disappear (e.g., when a method returns). This allows us to recover a more powerful permission. For example, $\mathsf{full} \Rrightarrow \frac{1}{2} \cdot \mathsf{share} \otimes \frac{1}{2} \cdot \mathsf{share} \Rrightarrow \mathsf{full}$.

3.2 Plural: Access permissions for Java

Our tool, Plural[2], is a plug-in to the Eclipse IDE that implements a previously developed type system [6, 3] as a static dataflow analysis for Java [7]. In the remainder of this section we show example annotations and explain how permissions are tracked and API implementations are verified.

Developer annotations. Developers use Java 5 annotations to specify method pre- and post-conditions with access permissions. Figure 2 shows the `Blocking_queue` specification with Plural annotations (compare to figure 1). We use `@States` to declare two parallel *state dimensions*, *protocol* and *structure*, which represent orthogonal parts of object state (the need for two dimensions in this example will be explained in the next section). Permission-named annotations on methods specify *borrowed* permissions for the receiver. Borrowed permissions are returned to the caller when the method returns. The optional attribute "value" specifies the dimension which the permission gives access to. For example, `close` borrows a full permission to operate only on the *protocol* dimension. Additionally, a required (or ensured) state must hold when the method is called (or returns), as illustrated by `close`. Boolean *state tests* such as `is_closed` can additionally be annotated with the state implied by each return value. Finally, we use `@Perm` annotations to declare permissions required and ensured by a method separately, such as in the constructor (which "produces" a receiver permission) or `enqueue_last_item` (which "consumes" a permission).

Permission tracking and local permission inference. Figure 3 shows a simple consumer thread that pulls items of a blocking queue. Plural is able to check whether this code respects the protocol declared for the `Blocking_queue` interface in figure 2. This program is buggy! Plural complains that the blocking queue referenced by `q` is not *STILLOPEN* when `dequeue` is called. The program seemingly just established that fact when entering the `while` loop, but because only a pure permission is used for testing the queue's state, Plural rightly assumes that other threads could have closed the queue in between the two calls to `q` in figure 2. Plural will no longer issue a warning after the addition of a `synchronized` block.

Notice that we use the same annotations for annotating method parameters in client code that we used for declaring API protocols in the previous section. Plural applies intra-procedural analysis and uses these annotations to track permissions across method calls. Conversely, Plural needs no annotations inside method bodies: based on the annotations provided, Plural infers how permissions flow through method bodies fully automatically. Since Plural is based on a dataflow analysis, it computes the fixed point of a loop without requiring the programmer to write a loop invariant. Local variables, variables of primitive type, and method parameters whose protocols we do not wish to check need not be annotated at all.

API implementation checking. Plural not only checks whether a client of an API follows the protocol required by that API, it can also check that the implementation

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

```
@Refine({
   @States(dim="structure", value={"STRUCTURESTATE"}),
   @States(dim="protocol", value={"CLOSED", "STILLOPEN"}),
   @States(value={"OPEN", "CLOSING"}, refined="STILLOPEN") })
class Blocking_queue {
   // fields omitted
   @Perm(ensures="unique(this) in OPEN,STRUCTURESTATE")
   Blocking_queue() { ... }

   @Share(value="structure")
   @Full(value="protocol", requires="OPEN", ensures="OPEN")
   void enqueue(Object o) { ... }

   @Share(value="structure")
   @Perm(requires="full(this, structure) in OPEN")
   void enqueue_final_item(Object o) { ... }

   @Share(value="structure")
   @Pure(value="protocol", requires="STILLOPEN")
   Object dequeue() { ... }

   @Pure(value="protocol")
   @TrueIndicates("CLOSED") @FalseIndicates("STILLOPEN")
   boolean is_closed() { ... }

   @Full(value="protocol", ensures="CLOSED")
   void close() { ... }
}
```

Fig. 2 Simplified `Blocking_queue` specification in Plural (using the typestates from figure 1)

```
public void consumerThread(
  @Share("structure") @Pure("protocol") Blocking_queue<String> q)
{
  while ( !q.is_closed() ) {
    String s = q.dequeue(); // Error! q may have been closed
    System.out.println("I received the message " + s);
  }
  // Thread continues...
}
```

Fig. 3 Simple `Blocking_queue` client with concurrency bug that is detected by Plural

of the protocol is consistent with its specification. The key abstraction for this is the *state invariant*, which we adapted from Fugue [12]. A state invariant associates a typestate of a class with a predicate over the fields of that class. In our approach, these predicates usually consist of access permissions for fields and look similar to the contents of `@Perm` annotations (see figure 2).

4 Reflections on Permissions

After six years of verifying object protocols with access permissions, we can reflect on Plural's position in the larger design space of typestate checkers, some of the ways in which the approach is successful, and other ways in which it is not. In this section we will reflect upon some of the more interesting things we have noticed during our experience using Plural.

4.1 Design Space

Typestate checking with Plural represents a compromise between many conflicting goals including expressiveness, the amount of developer input required, and the precision and scale at which Plural is able to perform typestate checking.

- *Expressiveness* here means the ability to express the rules that API designers would like API clients to follow. Plural specifications are by design not as expressive as behavioral specification languages like the JML [19], to keep developer input low and performance at a level acceptable for interactive use. On the other hand, we were repeatedly surprised that access permissions let us encode protocols we didn't think could be encoded with finite state machines. In particular, we found that Plural can express protocols involving multiple objects such as collections and iterators over them [6] which were previously not expressible in modular typestate checkers [22].
- *Developer input* is the amount of help Plural needs to do its job. Plural limits developer input to Java annotations for method parameters and instance fields (and infers permissions inside methods). We empirically found that about 2 annotations per method are sufficient [4]. This is far less than what full-fledged program verification requires [19]. On the other hand, we decided not to make Plural an inter-procedural analysis (like [22]). That allows Plural to check individual source files quickly, improves analysis precision (in particular for challenging protocols like iterators over collections), and lets Plural check library code for compliance to its advertised protocols [4].
- *Analysis precision* is the amount of false positives (spurious protocol violation warnings) and false negatives (missed actual protocol violations) reported by Plural. Plural is envisioned to work like a typechecker and hence sound [6, 3] so false negatives cannot occur. Empirically we found Plural's false positive rate to be less than 6 false positives per kloc in sample open-source programs [7].
- *Scalability* refers to Plural's ability to produce verification results for large programs. Plural performs an intra-procedural dataflow analysis, which can suffer in performance for large methods. On the other hand, dataflow analysis allows Plural to reduce developer input by inferring loop invariants etc. In practice, Plural checked sample open-source programs in less than 200 ms per method (with a

3.2 GHz CPU and 2 GM of RAM) [4]. Since methods are checked individually, Plural scales to programs with arbitrarily many methods.

As suggested above, typestate checking involves compromises. The following subsections explore some of the difficulties and benefits of our approach.

4.2 Difficulties

Even with the introduction of flexible aliasing permissions, the fact remains that verification in the face of aliasing can be quite difficult. One interesting, if unsurprising, observation is that the difficulty associated with verifying a piece of code is directly proportional to the permission types that are used. Code using immutable permissions is almost trivial to verify, as such objects do not change state, and their permissions can be freely duplicated. This nicely lines up with the statements made by those in the functional programming community, who have long argued that effect-free programs are easier to reason about. Our system provides such benefits, while still making the verification of imperative code possible when necessary. Next, unique permissions are quite easy to verify when the relevant objects are unaliased. Permissions give us the freedom to reason about these objects locally, even as side-effecting methods are called. At the other end of the spectrum are share and pure permissions. They can be quite difficult to use because of their limited guarantees. Both essentially say, "all bets are off," with respect to the behavior of an object.

In fact, the share permission is worthy of particular discussion since it was widely used throughout our case studies. In practice, typical methods in an object-oriented program consist of a series of method calls, and often little else! Unfortunately, such methods are poorly suited for verifying share permissions. In the Plural methodology, at any call-site, what we know about states of all the share permissions in the static context must be "forgotten," downgraded to the guaranteed state. This is because of the potential that such objects are modified through a different alias. In a method with numerous call sites, this means that the state of an object of share permission may not persist from the line where it is established to the line where it must be used. As discussed in the next section, state guarantees serve to make share and pure permissions more powerful, but when a state cannot be guaranteed, these permissions can be quite difficult to use.

It was intriguing to see that fractions, one of the most technically interesting pieces of our approach, were not needed terribly often. Fractions are useful for reassembling weak permission pieces together in order to recreate stronger permissions. However, in our experience, most objects in an object-oriented program are either designed to be aliased or unaliased. After initialization, their level of aliasedness does not typically change much, and if it does, correct performance of the program does not depend on reestablishing stronger permissions. One of the most important use cases for fractional permissions is in concurrent programs, where a number of threads take reference to a shared object, read from the object in parallel and then *join* together, so that the single remaining thread has unique permis-

sion. However, with the exception of a few scientific-style applications, such thread forking and joining is not typical in object-oriented programs. In our experience, programs rarely depend on all other references being dropped. This has important implications, because the theoretical and engineering machinery necessary to make fractions work is vast. Doing away with such machinery would leave a system that is more practicable.

4.3 Surprising Power

Fortunately, despite some of the issues we encountered, we found the access permissions approach to be quite powerful, including in some ways we found surprising. First, and as previously mentioned, we got an extraordinary amount of leverage out of state guarantees. Guarantees make the even the weakest permissions useful. Such a feature does not exist in other verification systems, in which very little can be done with arbitrarily aliased objects. Often, such permissions will be completely outside of the verification methodology, as in Fugue [12]. Such guarantees are only useful because of state hierarchies, a feature that is novel to our typechecking approach. Hierarchies allow simultaneous references to the same object with varrying levels of precision.

A recurring theme in our case studies was an interplay between states and permissions in verification. In our initial conception, we developed a hierarchical system of states expressive enough to model many of the most intricate protocols we observed in the Java standard library. Then, in order to verify such models in the face of aliasing, the system of access permissions was developed. In other words, the specification of protocols and aliasing were conceived as two orthogonal issues. But in practice, there was an interesting interplay between the two. Features designed for the specification of state machines helped in alias-control, and vice-versa.

As an example, consider the specification of the blocking queue in Figure 2. This class is specified with a structure dimension, into which the underlying linked list is mapped. This allows both producer and consumer threads, each of which have a share permission to this dimension, to modify the linked list, while at the same time, only the producer thread has modifying rights to the protocol dimension, which contains the queue's actual protocol. Dimensions were initially conceived for use in classes that define multiple orthogonal state machines. In this case, however, the structure dimension defines a protocol that is completely uninteresting; it consists of a single state. It is, rather, because we can map fields into dimensions, and hand out permissions to those dimensions independently, that the queue is given a second dimension at all! Thus, in this example, dimensions serve to naturally encode the intuition that anyone can modify the queue, but the producer is in charge of determining when it can be closed; without dimensions, verifying the queue would be impossible in our system.

Dimensions have proved themselves to be quite a powerful abstraction. In numerous cases we found it useful to store a permission to an object inside the object itself.

In order to do this, a permission to one dimension is stored inside another dimension. Consider the `enqueue_final_item` method of the `Blocking_queue` class. When this method is called, the producer is conceptually stating that no further items will be enqueued, but that the consumers are free to dequeue all of the items waiting in the queue. When the last item is dequeued, that consumer is responsible for actually closing the queue. The question arises, "how can a consumer thread call the `close` method when it requires full permission and consumers have only pure?" The answer is that when the producer calls the method to enqueue the final item, it must forfeit its full permission. This permission is then stored in the structure dimension, and is used by the consumer thread when the last item is dequeued. This idiom can be thought of as encoding the intuition that permission to do something—closing the queue in this case—is carried through the queue from the producer to the consumer along with the last element.

As another example of the interplay between permissions and states, consider that programs often may pass through various "phases" of aliasing. For example, in one phase an object may be completely unaliased as it is being initialized. Later on, aliases may be created in an unrestricted fashion. Such phases do not exist by coincidence. Developers use these phases to reason about their own programs, to tell themselves what must be true at any given point in the program. It turns out that these phases nicely correspond to abstract states in an object protocol, and permissions can be used in a natural way to express design intent and to aid in verification even in situations where objects do not define protocols in the typical sense. Many times we used such a pattern: an object referencing one or more fields would be constructed in an "uninitialized" state. The invariants associated with this state indicate that the fields are unaliased, perhaps also in their own initialization states. Later on, after all fields have been set up, and the object is in the "initialized," it is legal to call its getter methods, which return aliases to the internal objects.

5 Conclusion and Ongoing Research

Verifying protocols is important due to their ubiquity in modern software development, yet it is difficult in the presence of aliasing. The Plural system has achieved some success by leveraging access permissions, which express design intent with respect to how an object is aliased and how it is used through those aliases. The different features of access permissions were designed to capture the natural forms of reasoning that engineers are using anyway to think about protocols. We have found that the concepts in access permissions are complementary in surprising ways, allowing fairly simple mechanisms to verify quite complex examples.

A number of current research projects have been inspired by Plural and related permissions systems. These projects build on the themes of natural reasoning and design intent, and promise to increase the impact of permissions and extend their benefits into new application areas:

Permission-Based Type Systems. What benefits might there be from designing the type system of a programming language around Plural-like permissions? The Plaid language is an attempt to explore this research question [1]. In Plaid, not only can the interface of an object change (as in typestate), but the representation (fields) and behavior (method implementations) can change as well. Closer integration into the language appears to smooth over some rough edges of Plural (such as syntax and the handling of inheritance) while reducing functionality overlaps (such as duplicate parameterization mechanisms for permissions and Java types). It also opens intriguing new possibilities, such as the ability to test permissions with a run-time cast that can verify, for example, that a reference to an object is really unique.

In looking at permission-based type systems, we have also begun to explore the idea of merging the permission kind and typestate parts of a specification into a single abstraction, and providing customized rules for how different permission abstractions can be split and merged [21]. This design may eventually provide more flexibility compared to the 5 fixed permission kinds in Plural.

Impact Analysis. An innovative use of Plural that does not involve API protocols at all is impact analysis [25], where it can be used to understand implicit state dependencies between components (in contrast to dependencies due to explicit messages being passed between components). When changing a component that has read-only access to shared state then the change cannot impact other component with access to the same state. Plural's distinction between read-only and read-write references allows making this determination, increasing impact analysis precision.

Program Verification. Plural employs fractional permissions [9] specifically for typestate reasoning. But fractions are more broadly useful for reasoning about program behavior under aliasing [8]. More expressive program verification tools that incorporate fractional permissions have since been developed [17, 20]. Plural and these tools show that fractional permissions—which were originally proposed for reasoning about data races—are not only useful for reasoning about concurrent programs but also well-suited for reasoning about sequential programs. In particular, permissions provide new ways to express how different objects collaborate [6].

Acknowledgements We would like to thank the many people who have given us feedback and encouragement over the years, including John Boyland, Frank Pfenning, and Ciera Jaspan. We are also very thankful to Matthew Dwyer for giving us the opportunity to present this article. This work was supported in part by DARPA grant #HR0011-0710019 and NSF grant CCF-0811592. While at Carnegie Mellon University, the second author was supported by a National Science Foundation Graduate Research Fellowship (DGE-0234630).

References

1. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Onward!*, 2009.

2. N. E. Beckman. *Types for Correct Concurrent API Usage*. PhD thesis, technical report CMU-ISR-10-131. Carnegie Mellon University, Dec. 2010.
3. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2008.
4. K. Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, technical report CMU-ISR-09-108. Carnegie Mellon University, Apr. 2009.
5. K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Sept. 2005.
6. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2007.
7. K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, July 2009.
8. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Principles of Programming Languages*, Jan. 2005.
9. J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, 2003.
10. C. Jaspan. *Proper Plugin Protocols*, 2010. Carnegie Mellon University Thesis Proposal.
11. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation*, 2001.
12. R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, 2004.
13. L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, Sept. 1976.
14. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
15. J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Object-Oriented Programming, Systems, Languages & Applications*, 1991.
16. R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy. *The Turing Language: Design and Definition*. Prentice-Hall, 1988.
17. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.
18. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Programming Languages and Systems*, 5(4):596–619, 1983.
19. 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.
20. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, Mar. 2009.
21. F. Militão, J. Aldrich, and L. Caires. Aliasing control with view-based typestate. In *Formal Techniques for Java-like Programs*, 2010.
22. N. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2008.
23. J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*, 1998.
24. K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. Software Engineering*, 16(3):268–280, Mar. 1990.
25. D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic. Helios: Impact analysis for event-based systems. Technical Report USC-CSSE-2009-517, University of Southern California, Nov. 2009.
26. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. on Software Engineering*, 12(1):157–171, 1986.
27. P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.