Modular Typestate Checking of Aliased Objects

Kevin Bierhoff Jonathan Aldrich

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

Abstract

Objects often define usage protocols that clients must follow in order for these objects to work properly. Aliasing makes it notoriously difficult to check whether clients and implementations are compliant with such protocols. Accordingly, existing approaches either operate globally or severely restrict aliasing.

We have developed a sound modular protocol checking approach, based on typestates, that allows a great deal of flexibility in aliasing while guaranteeing the absence of protocol violations at runtime. The main technical contribution is a novel abstraction, *access permissions*, that combines typestate and object aliasing information. In our methodology, developers express their protocol design intent through annotations based on access permissions. Our checking approach then tracks permissions through method implementations. For each object reference the checker keeps track of the degree of possible aliasing and is appropriately conservative in reasoning about that reference. This helps developers account for object manipulations that may occur through aliases. The checking approach handles inheritance in a novel way, giving subclasses more flexibility in method overriding. Case studies on Java iterators and streams provide evidence that access permissions can model realistic protocols, and protocol checking based on access permissions can be used to reason precisely about the protocols that arise in practice.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification.

Keywords Typestates, aliasing, permissions, linear logic, behavioral subtyping.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction

In object-oriented software, objects often define *usage protocols* that clients must follow in order for these objects to work properly. Protocols essentially define legal sequences of method calls. In conventional object-oriented languages, developers have three ways of finding out about protocols: reading informal documentation, receiving runtime exceptions that indicate protocol violations, or observing incorrect program behavior as a result of protocol violations that broke internal invariants.

It is the goal of this work to help developers follow protocols while they write code as well as to allow them to correctly and concisely document protocols for their code. We build on our previous work on leveraging *typestates* [34] for lightweight object protocol specification [4]. Our protocols are state machines that are reminiscent of Statecharts [20].

Aliasing, i.e. the existence of multiple references to the same object, is a significant complication in checking whether clients observe a protocol: a client does not necessarily know whether its reference to an object is the only reference that is active at a particular execution point. This also makes it difficult to check whether a class implements its specified protocol because reentrant callbacks through aliases can again lead to unexpected state changes.

Existing protocol checking approaches fall into two categories. They either operate globally, i.e. check an entire code base at once, or severely restrict aliasing. Global analyses typically account for aliasing but they are not suitable for interactive use during development. Moreover, they do not check whether a declared protocol is implemented correctly, a crucial requirement in object-oriented software where any class might have a protocol of its own.

Modular protocol checkers, like Fugue [12], the first sound modular typestate checker for an object-oriented language, better support developers while they write code: like a typechecker, they check each method separately for protocol violations while assuming the rest of the system to behave as specified. The trade-off, unfortunately, has been that modular checkers require code to follow pre-defined patterns of aliasing. Once a program leaves the realm of supported aliasing, any further state changes are forbidden. Generally speaking, state changes are only allowed where the checker is aware of *all* references to the changing object.

This approach has serious drawbacks. First, many examples of realistic code might be excluded. Moreover, from a developer's point of view, the boundaries of what a checker supports are hard to predict and they might not fit with the best implementation strategy for a particular problem. Finally, aliasing restrictions arguably leave developers alone just when they have the most trouble in reasoning about their code, namely, in the presence of subtle aliasing.

This paper proposes a sound modular typestate checking approach for Java-like object-oriented languages that allows a great deal of flexibility in aliasing. For each reference, it tracks the degree of possible aliasing, and is appropriately conservative in reasoning about that reference. This helps developers account for object manipulations that may occur through aliases. High precision in tracking effects of possible aliases together with systematic support for *dynamic state tests*, i.e. runtime tests on the state of objects, make this approach feasible. Our approach helped expose a way of breaking an internal invariant that causes a commonly used Java standard library class, `java.io.BufferedInputStream`, to access an array outside its bounds. Contributions of this paper include the following.

- Our main technical contribution is a novel abstraction, called *access permissions*, that combines typestate with aliasing information about objects. Developers use access permissions to express the *design intent* of their protocols in annotations on methods and classes. Our modular checking approach verifies that implementations follow this design intent.

  Access permissions systematically capture different patterns of aliasing (figure 1). A permission tracks (a) how a reference is allowed to read and/or modify the referenced object, (b) how the object might be accessed through other references, and (c) what is currently known about the object's typestate.

- In particular, our full and pure permissions [3] capture the situation where one reference has exclusive write access to an object (a full permission) while other references are only allowed to read from the same object (using pure permissions). Read-only access through pure permissions is intuitively harmless but has to our knowledge not been exploited in existing modular protocol checkers.

- In order to increase precision of access permissions, we include two additional novel features, which make *weak permissions* more useful than in existing work. We call permissions "weak" if the referenced object can potentially be modified through other permissions.

  - *Temporary state information* can be associated with weak permissions. Our checking approach makes sure that temporary state information is "forgotten" when it becomes outdated.
  - Permissions can be confined to a particular part of the referenced object's state. This allows separate permissions to independent parts of the same object. It also implies a *state guarantee* even for weak permissions, i.e. a guarantee that the referenced object will not leave a certain state.

Access through other permissions	Current permission has ... Read/write access	Read-only access
None	unique [6]	–
Read-only	full [3]	immutable [6]
Read/write	share [12]	pure [3]

Figure 1. Access permission taxonomy

- We handle inheritance in a novel way, giving subclasses more flexibility in method overriding. This is necessary for handling realistic examples of inheritance such as Java's `BufferedInputStream` (details in section 3.2).

- We validated our approach with two case studies, iterators (section 2) and streams (section 3) from Sun's Java standard library implementation. These case studies provide evidence that access permissions can model realistic protocols, and protocol checking based on access permissions can be used to reason precisely about the protocols that arise in practice.

A more complete evaluation of our approach is beyond the scope of this paper, which focuses on fully presenting our checking technique. The evaluation does establish that our—compared to full-fledged program verification systems [26, 2]—relatively simple approach can verify code idioms and find errors that no other decidable modular system can. The case studies reflect actual Java standard library protocols and, as far as we can tell, cannot be handled by any existing modular protocol verification system.

The following two sections introduce access permissions and verification approach with examples from our case studies before sections 4 and 5 give a formal account of our approach. Section 6 compares our approach to related work.

2. Read-Only Iterators

This section illustrates basic protocol specification and verification using our approach based on a previous case study on Java *iterators* [3]. Iterators follow a straightforward protocol but define complicated aliasing restrictions that are easily violated by developers. They are therefore a good vehicle to introduce our approach to handling aliasing in protocol verification. Iterators as presented here cannot be handled by existing modular typestate checkers due to their aliasing restrictions.

2.1 Specification Goals

The specification presented in this section models the `Iterator` interface defined in the Java standard library. For the sake of brevity we focus on *read-only* iterators, i.e. iterators that cannot modify the collection on which they iterate. We will refer to read-only iterators simply as "iterators" and qualify full Java iterators as "modifying iterators". In earlier work we showed how to capture full Java iterators [3]. Goals of the presented specification include the following.

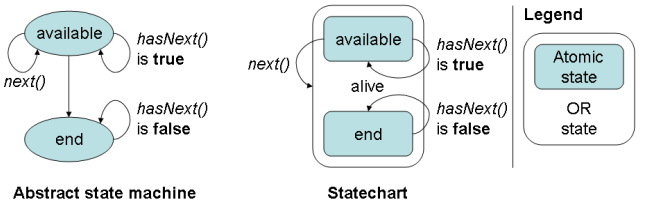

Figure 2. Read-only iterator state machine protocol

- Capture the usage protocol of Java iterators.
- Allow creating an arbitrary number of iterators over collections.
- Invalidate iterators before modification of the iterated collection.

2.2 State Machine Protocol

An iterator returns all elements of an underlying *collection* one by one. Collections in the Java standard library are lists or sets of objects. Their interface includes methods to add objects, remove objects, and test whether an object is part of the collection. The interface also defines a method `iterator` that creates a new iterator over the collection. Repeatedly calling `next` on an iterator returns each object contained in the iterated collection exactly once. The method `hasNext` determines whether another object is available or the iteration reached its end. It is illegal to call `next` once `hasNext` returns `false`. Figure 2 illustrates this protocol as a simple state machine.

Notice that `hasNext` is legal in both states but does not change state. We call `hasNext` a *dynamic state test*: its return value indicates what state the iterator is currently in. The next section will show how this protocol can be specified.

2.3 Iterator Interface Specification

States Through Refinement. We call the set of possible states of an object its *state space* and define it as part of the object's interface. As suggested above, we can model the iterator state space with two states, available and end. In our approach, states are introduced by *refinement* of an existing state. State refinement corresponds to OR-states in Statecharts [20] and puts states into a tree hierarchy.

State refinement allows interfaces to, at the same time, *inherit* their supertypes' state spaces, define additional (more fine-grained) states, and be properly *substitutable* as subtypes of extended interfaces [4]. Refinement guarantees that all new states defined in a subtype correspond to a state inherited from the supertype. States form a hierarchy rooted in a state alive defined in the root type `Object`. Iterators therefore define their state space as follows.

```
states available, end refine alive;
```

Typestates do *not* correspond to fields in a class. They describe an object's state of execution abstractly and information about fields can be *tied* to typestates using state invariants (see section 3.1).

Access Permissions Capture Design Intent. Iterators have only two methods, but these have very different behavior. While `next` can *change* the iterator's state, `hasNext` only *tests* the iterator's state. And even when a call to `next` does not change the iterator's state, it still advances the iterator to the next object in the sequence. `hasNext`, on the other hand, is *pure*: it does not modify the iterator at all.

We use a novel abstraction, *access permissions* ("permissions" for short), to capture this *design intent* as part of the iterator's protocol. Permissions are associated with object references and govern how objects can be accessed through a given reference [7]. For `next` and `hasNext` we only need two kinds of permissions; more kinds of permissions will be introduced later.

- full permissions grant read/write access to the referenced object *and guarantee that no other reference has read/write access* to the same object.
- pure permissions grant read-only access the the referenced object but *assume that other permissions could modify* the object.

A distinguished full permission can co-exist with an arbitrary number of pure permissions to the same object. This property will be enforced when verifying protocol compliance. In a specification we write *perm*(x) for a permission to an object referenced by x, where *perm* is one of the permission kinds. Access permissions carry state information about the referenced object. For example, "full(*this*) `in` available" represents a full permission for an object (*this*) that is in the available state.

Linear Logic Specifications. Methods can be specified with a *state transition* that describes how method parameters change state during method execution. We previously argued that existing typestate verification approaches are limited in their ability to express realistic state transitions [4] and proposed to capture method behavior more precisely with logical expressions.

Access permissions represent resources that have to be consumed upon usage—otherwise permissions could be freely duplicated, possibly violating other permissions' assumptions. Therefore, we base our specifications on linear logic [18]. Pre- and post-conditions are separated with a linear implication ($\multimap$) and use conjunction ($\otimes$) and disjunction ($\oplus$).[1] In certain cases, internal choice ($\&$, also called additive conjunction) has been useful [3]. These connectives represent the decidable multiplicative-additive fragment of linear logic (MALL).

Iterators illustrate that state transitions are often nondeterministic. For `next`, we can use an *imprecise* postcondition and specify `next` so that it requires a full permission in state available and returns the full permission in the

[1] "Tensor" ($\otimes$) corresponds to conjunction, "alternative" ($\oplus$) to disjunction, and "lollie" ($\multimap$) to implication in conventional logic. The key difference is that linear logic treats known facts as resources that are consumed when proving another fact. This fits well with our intuition of permissions as resources that give access to objects.

alive state. In a Statechart, this corresponds to transitioning to a state that contains substates (figure 2).

$$\mathsf{full}(\mathit{this})\ \texttt{in}\ \mathsf{available} \multimap \mathsf{full}(\mathit{this})\ \texttt{in}\ \mathsf{alive}$$

Dynamic state tests (like `hasNext`) require relating the (Boolean) method result to the state of the tested object (usually the receiver). A disjunction of conjunctions expresses the two possible outcomes of `hasNext` (figure 4) where each conjunction relates a possible method result to the corresponding receiver state. (We adopt the convention that $\multimap$ binds weaker than $\otimes$ and $\oplus$.)

$$\begin{aligned}\mathsf{pure}(\mathit{this}) \multimap\ & (\mathit{result} = \texttt{true} \otimes \mathsf{pure}(\mathit{this})\ \texttt{in}\ \mathsf{available}) \\ & \oplus\ (\mathit{result} = \texttt{false} \otimes \mathsf{pure}(\mathit{this})\ \texttt{in}\ \mathsf{end})\end{aligned}$$

These specifications enforce the characteristic `hasNext` / `next` call pairing: `hasNext` determines the iterator's current state. If it returns `true` then it is legal to call `next`. The iterator is in an unknown state after `next` returns, and another `hasNext` call determines the iterator's new state.

2.4 Creating and Disposing Iterators

Multiple (independent) iterators are permitted for a single collection at the same time. However, the collection must not be modified while iteration is in progress. Standard implementations try to detect such situations of *concurrent modification* on a best-effort basis. But, ultimately, Java programmers have to make sure on their own that collections are not modified while iterated. (Note that "concurrent" modifications often occur in single-threaded programs [32].)

This section shows how the aliasing constraints between iterators and its collection can be handled. As we will see, this problem is largely orthogonal to specifying the relatively simple protocol for for individual iterators that was discussed in the previous section.

Immutable Access Prevents Concurrent Modification. Access permissions can guarantee the absence of concurrent modification. The key observation is that when an iterator is created it stores a reference to the iterated collection in one of its fields. This reference should be associated with a permission that guarantees the collection's *immutability* while iteration is in progress. We include two previously proposed permissions [6] into our system in order to properly specify collections.

- immutable permissions grant read-only access to the referenced object *and guarantee that no reference has read/write access* to the same object.
- unique permissions grant read/write access *and guarantee that no other reference has* any *access* to the object.

Thus immutable permissions *cannot* co-exist with full permissions to the same object. We can specify the collection's `iterator` method using these permissions as follows. Notice how it *consumes* or *captures* the incoming receiver permission and returns an initial unique permission to a fresh iterator object.

```
public class Collection {
  Iterator iterator() : immutable(this) —o unique(result)
}
```

It turns out that this specification precisely captures Sun's Java standard library implementation of iterators: Iterators are realized as inner classes that implicitly reference the collection they iterate.

Permission Splitting. How can we track permissions? Consider a client such as the one in figure 3. It gets a unique permission when first creating a collection. Then it creates an iterator which captures an immutable permission to the collection. However, the client later needs more immutable permissions to create additional iterators. Thus while a unique permission is intuitively stronger than an immutable permission we cannot just coerce the client's unique permission to an immutable permission and pass it to `iterator`: it would get captured by the newly created iterator, leaving the client with no permission to the collection at all.

In order to avoid this problem we use *permission splitting* in our verification approach. Before method calls we split the original permission into two, one of which is retained by the caller. Permissions are split so that their assumptions are not violated. In particular, we never duplicate a full or unique permission and make sure that no full permission co-exists with an immutable permission to the same object. Some of the legal splits are the following.

$$\begin{aligned}\mathsf{unique}(x) &\Rrightarrow \mathsf{full}(x) \otimes \mathsf{pure}(x) \\ \mathsf{full}(x) &\Rrightarrow \mathsf{immutable}(x) \otimes \mathsf{immutable}(x) \\ \mathsf{immutable}(x) &\Rrightarrow \mathsf{immutable}(x) \otimes \mathsf{immutable}(x) \\ \mathsf{immutable}(x) &\Rrightarrow \mathsf{immutable}(x) \otimes \mathsf{pure}(x)\end{aligned}$$

They allow the example client in figure 3 to retain an immutable permission when creating iterators, permitting multiple iterators and reading the collection directly at the same time.

Permission Joining Recovers Modifying Access. When splitting a full permission to a collection into immutable

```
Collection c = new ...
Iterator it = c.iterator();          // legal
while(it.hasNext() && ...) {         // legal
    Object o = it.next();            // legal
    Iterator it2 = c.iterator();     // legal
    while(it2.hasNext()) {           // legal
        Object o2 = it2.next();      // legal
        ... }
}
if(it.hasNext() && c.size() == 3) {  // legal
    c.remove(it.next());             // legal
    if(it.hasNext()) ... }           // ILLEGAL
Iterator it3 = c.iterator();         // legal
```

Figure 3. A simple `Iterator` client

```
interface Iterator<c : Collection, k : Fract> {
  states available, end refine alive

  boolean hasNext() :
      pure(this) ⊸ (result = true ⊗ pure(this) in available))
                 ⊕ (result = false ⊗ pure(this) in end)
  Object next() :
      full(this) in available ⊸ full(this)
  void finalize() :
      unique(this) ⊸ immutable(c, k)
}

interface Collection {
  void add(Object o) : full(this) ⊸ full(this)
  int size() : pure(this) ⊸ result ≥ 0 ⊗ pure(this)
  // remove(), contains() etc. similar

  Iterator<this, k> iterator() :
      immutable(this, k) ⊸ unique(result)
}
```

Figure 4. Read-only `Iterator` and partial `Collection` interface specification

permissions we lose the ability to modify the collection. Intuitively, we would like to reverse permission splits to regain the ability to modify the collection.

Such *permission joining* can be allowed if we introduce the notion of fractions [6]. Essentially, fractions keep track of how often a permission was split. This later allows joining permissions (with known fractions) by putting together their fractions. A unique permission by definition holds a *full fraction* that is represented by one (1). We will capture fractions as part of our permissions and write $(perm)(x, k)$ for a given permission with fraction k. We usually do not care about the exact fraction and therefore implicitly quantify over all fractions. If a fraction does not change we often will omit it. Fractions allow us to define splitting and joining rules as follows.

$$\begin{aligned}
\mathsf{unique}(x,1) &\Longleftrightarrow \mathsf{full}(x,1/2) \otimes \mathsf{pure}(x,1/2)\\
\mathsf{full}(x,k) &\Longleftrightarrow \mathsf{immutable}(x,k/2) \otimes \mathsf{immutable}(x,k/2)\\
\mathsf{immutable}(x,k) &\Longleftrightarrow \mathsf{immutable}(x,k/2) \otimes \mathsf{immutable}(x,k/2)\\
\mathsf{immutable}(x,k) &\Longleftrightarrow \mathsf{immutable}(x,k/2) \otimes \mathsf{pure}(x,k/2)
\end{aligned}$$

For example, we can split $\mathsf{full}(it, 1/2)$ into $\mathsf{full}(it, 1/4) \otimes \mathsf{pure}(it, 1/4)$ and recombine them. Such reasoning lets our iterator client recover a unique iterator permission after each call into the iterator.

Recovering Collection Permissions. Iterators are created by trading a collection permission for a unique iterator permission. We essentially allow the opposite trade as well in order to modify a previously iterated collection again: We can safely consume a unique iterator permission and recover the permissions to its fields because no reference will be able to access the iterator anymore. A simple live variable analysis can identify when variables with unique permissions are no longer used. (As a side effect, a permission-based approach therefore allows identifying dead objects.)

```
Collection c = new ...                                  unique(c)
Iterator it<c, 1/2> = c.iterator();
                                  immutable(c, 1/2) ⊗ unique(it)
while(it.hasNext() && ...) {
                  immutable(c, 1/2) ⊗ unique(it) in available
    Object o = it.next();
                                  immutable(c, 1/2) ⊗ unique(it)
    Iterator it2<c, 1/4> = c.iterator();
                 immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2)
    while(it2.hasNext()) {
     immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2) in available
        Object o2 = it2.next();
                 immutable(c, 1/4) ⊗ unique(it) ⊗ unique(it2)
        ... } // it2 dies
}                                 immutable(c, 1/2) ⊗ unique(it)
if(it.hasNext() && c.size() == 3) {
                  immutable(c, 1/2) ⊗ unique(it) in available
    c.remove(it.next()); // it dies after next()
                               unique(c) and no permission for it
    if(it.hasNext()) ... }                        // ILLEGAL
// it definitely dead                                   unique(c)
Iterator it3<c, 1/2> = c.iterator();
                                  immutable(c, 1/2) ⊗ unique(it3)
```

Figure 5. Verifying a simple `Iterator` client

For lack of a more suitable location, we annotate the `finalize` method to indicate what happens when an iterator is no longer usable. And in order to re-establish *exactly* the permission that was originally passed to the iterator we parameterize `Iterator` objects with the collection permission's fraction. The `finalize` specification can then release the captured collection permission from dead iterators. The complete specification for iterators and a partial collection specification are summarized in figure 4.

2.5 Client Verification

Figure 5 illustrates how our client from figure 3 can be verified by tracking permissions and splitting/joining them as necessary. After each line of code we show the current set of permissions on the right-hand side of the figure. We recover collection permissions from dead iterators as soon as possible. This lets us verify the entire example client. We correctly identify the seeded protocol violation.

2.6 Summary

We presented a specification of read-only iterators that prevents concurrent collection modification. To this end it associates collections and iterators with *access permissions*, defines a simple state machine to capture the iterator usage protocol, and tracks permission information using a decidable fragment of linear logic. Our logic-based specifications can relate objects to precisely specify method behavior in terms of typestates and support reasoning about dynamic tests.

3. Java Stream Implementations

I/O protocols are common examples for typestate-based protocol enforcement approaches [11, 12, 4]. This section sum-

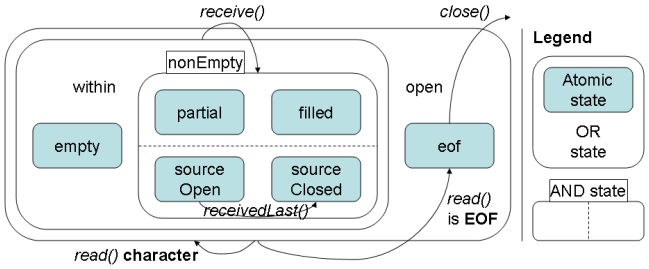

Figure 6. `PipedInputStream`'s state space (inside `open`)

marizes a case study in applying our approach to Java *character streams* and in particular *stream pipes* and *buffered input streams*. The section focuses on *implementation verification* of stream classes, which—to our knowledge—has not been attempted with typestates before. Implementation verification generalizes techniques shown in the previous section for client verification.

3.1 Stream Pipes

Pipes are commonly used in operating system shells to forward output from one process to another process. Pipes carry alphanumeric *characters* for a source to a sink. The Java I/O library includes a pair of classes, `PipedOutputStream` and `PipedInputStream`, that offers this functionality inside Java applications. This section provides a specification for Java pipes and shows how the classes implementing pipes in the Java standard library can be checked using our approach.

Informal Pipe Contract. In a nutshell, Java pipes work as follows: A character-producing "writer" `writes` characters into a `PipedOutputStream` (the "source") that forwards them to a connected `PipedInputStream` (the "sink") from which a "reader" can `read` them. The source forwards characters to the sink using the internal method `receive`. The writer calls `close` on the source when it is done, causing the source to call `receivedLast` on the sink (figure 7).

The sink caches received characters in a circular buffer. Calling `read` on the sink removes a character from the buffer (figure 8). Eventually the sink will indicate, using an *end of file token* (EOF, -1 in Java), that no more characters can be read. At this point the reader can safely close the sink. Closing the sink before EOF was read is unsafe because the writer may still be active.

The pipe classes in Sun's standard library implementation have built-in runtime checks that throw exceptions in the following error cases: (1) closing the sink before the source, (2) writing to a closed source or pushing characters to the sink after the source was closed, and (3) reading from a closed sink. The specification we present here makes these error cases impossible.

State Space with Dimensions. The *source protocol* can be modeled with three states raw, open, and closed. raw indicates that the source is not connected to a sink yet. For technical reasons that are discussed below, we refine open into ready and sending. The writer will always find the source in state ready.

For the *sink protocol* we again distinguish open and closed. A refinement of open helps capturing `read`'s protocol: The sink is within as long as `read` returns characters; the eof state is reached when `read` returns the EOF token. While within, we keep track of the sink's buffer being empty or nonEmpty. We further refine nonEmpty into partial and filled, the latter corresponding to a full buffer.

At the same time, however, we would like to track whether the source was closed, i.e., whether `receivedLast` was called. We previously proposed *state dimensions* to address such separate concerns (here, the buffer filling and the source state) [4] with states that are independent from each other. State dimensions correspond to AND-states in Statecharts [20].

We can simply refine nonEmpty *twice, along different dimensions*. We call the states for the second dimension sourceOpen and sourceClosed with the obvious semantics. Note that we only need the additional source dimension while the buffer is nonEmpty; the source is by definition open (closed) in the empty (eof) state.[2] To better visualize the sink's state space, figure 6 summarizes it as a Statechart.

Shared Modifying Access. Protocols for source and sink are formalized in figures 7 and 8 with specifications that work similar to the iterator example in the last section. However, the sink is conceptually modified through two distinct references, one held by the source and one held by the reader. In order to capture this, we introduce our last permission.

- share permissions grant read/write access to the referenced object but *assume that other permissions have read/write access as well*.

Conventional programming languages effectively always use share permissions for mutable state. Interestingly, share permissions are split and joined exactly like immutable permissions. Since share and immutable permissions cannot coexist, our rules force a commitment to either one when initially splitting a full permission.

$$\begin{aligned} \mathsf{full}(x,k) &\Longleftrightarrow \mathsf{share}(x,k/2) \otimes \mathsf{share}(x,k/2) \\ \mathsf{share}(x,k) &\Longleftrightarrow \mathsf{share}(x,k/2) \otimes \mathsf{share}(x,k/2) \\ \mathsf{share}(x,k) &\Longleftrightarrow \mathsf{share}(x,k/2) \otimes \mathsf{pure}(x,k/2) \end{aligned}$$

State Guarantees. We notice that most modifying methods cannot change a stream's state arbitrarily. For example, `read` and `receive` will never leave the open state and they cannot tolerate other permission to leave open.

We make this idea part of our access permissions. We include another parameter into permissions that specifies a *state guarantee*, i.e. a state that cannot be left even by modifying permissions. Thus a state guarantee (also called the permission's *root*) corresponds to an "area" in a Statechart that cannot be left. As an example, we can write the permis-

[2] This is only *one* way of specifying the sink. It has the advantage that readers need not concern themselves with the internal communication between source and sink.

```
public class PipedOutputStream {
  states raw, open, closed refine alive;
  states ready, sending refine open;

  raw := sink = null
  ready := half(sink, open)
  sending := sink ≠ null
  closed := sink ≠ null

  private PipedInputStream sink;

  public PipedOutputStream() :
    1 ⊸ unique(this) in raw  { }

  void connect(PipedInputStream snk) :
    full(this) in raw ⊗ half(snk, open) ⊸
      full(this) in ready
  { sink = snk;                    store permission in field
  }                                        full(this) in open

  public void write(int b) :
    full(this, open) in ready ⊗ b ≥ 0 ⊸ full(this, open) in ready
  {                              half(sink, open) from invariant
    sink.receive(b);                 returns half(sink, open)
  }                                  full(this, open) in ready

  public void close() :
    full(this) in ready ⊸ full(this) in closed
  {                              half(sink, open) from invariant
    sink.receivedLast();           consumes half(sink, open)
  }                                        full(this) in closed
}
```

Figure 7. Java `PipedOutputStream` (simplified)

sion needed for `read` as share(*this*, open). Without an explicit state guarantee, only alive is guaranteed (this is what we did for iterators).

State guarantees turn out to be crucial in making share and pure permissions useful because they guarantee a state even in the face of possible changes to the referenced object through other permissions. Moreover, if we combine them with state dimensions we get independent permissions for orthogonal object aspects that, e.g., let us elegantly model modifying iterators [3].

Explicit Fractions for Temporary Heap Sharing. When specifying the sink methods used by the source (`receive` and `receivedLast`) we have to ensure that the source can no longer call the sink after `receivedLast` so the sink can be safely closed. Moreover, in order to close the sink, we need to restore a permission rooted in alive. Thus the two share permissions for the sink have to be joined in such a way that there are definitely no other permissions relying on open (such permissions, e.g., could have been split off of one of the share permissions).

We extend the notion of fractions to accomplish this task. We use fractions to track, *for each state separately*, how many permissions rely on it. What we get is a *fraction function* that maps guaranteed states (i.e. the permission's

```
class PipedInputStream {
  stream = open, closed refines alive;
  position = within, eof refines open;
  buffer = empty, nonEmpty refines within;
  filling = partial, filled refines nonEmpty;
  source = sourceOpen, sourceClosed refines nonEmpty;

  empty := in ≤ 0 ⊗ closedByWriter = false
  partial := in ≥ 0 ⊗ in ≠ out
  filled := in = out
  sourceOpen := closedByWriter = false
  sourceClosed := closedByWriter ⊗ half(this, open)
  eof := in ≤ 0 ⊗ closedByWriter ⊗ half(this, open)

  private boolean closedByWriter = false;
  private volatile boolean closedByReader = false;
  private byte buffer[] = new byte[1024];
  private int in = -1, out = 0;

  public PipedInputStream(PipedOutputStream src) :
    full(src) in raw ⊸ half(this, open) ⊗ full(src) in open
  {  unique(this) in open ⇛ half(this, open) ⊗ half(this, open)
    src.connect(this);          consumes one half(this, open)
  }                          half(this, open) ⊗ full(src) in open

  synchronized void receive(int b) :
    half(this, open) ⊗ b ≥ 0 ⊸ half(this, open) in nonEmpty
  { // standard implementation checks if pipe intact
    while(in == out)                  half(this, open) in filled
      ... // wait a second
                              half(this, open) in empty ⊕ partial
    if(in < 0) { in = 0; out = 0; }
    buffer[in++] (byte)(b & 0xFF);
    if(in >= buffer.length) in = 0;
  }                                   half(this, open) in partial

  synchronized void receivedLast() :
    half(this, open) ⊸ 1
  { closedByWriter = true; }      this is now sourceClosed

  public synchronized int read() :
    share(this, open) ⊸ (result ≥ 0 ⊗ share(this, open))
                      ⊕ (result = −1 ⊗ share(this, open) in eof)
  { ... } // analogous to receive()

  public synchronized void close() :
    half(this, open) in eof ⊸ unique(this) in closed
  {     half(this, open) from eof invariant ⇛ unique(this, open)
    closedByReader = true;
    in = -1;
  }
}
```

Figure 8. Java `PipedInputStream` (simplified)

root and its super-states) to fractions. For example, if we split an initial unique permission for a `PipedInputStream` into two share permissions guaranteeing open then these permissions rely on open and alive with a $1/2$ fraction each. (Iterator permissions root in alive and their fraction functions map alive to the given fraction.)

In order to close the sink, we have to make sure that there are *exactly* two share permissions relying on open. Fraction functions make this requirement precise. For readability, we use the abbreviation half in figure 8 that stands for the following permission.

$$\mathsf{half}(x, \mathsf{open}) \equiv \mathsf{share}(x, \mathsf{open}, \{\mathsf{alive} \mapsto 1/2, \mathsf{open} \mapsto 1/2\})$$

By adding fractions and moving the state guarantee up in the state hierarchy, the initial permission for the sink, $\mathsf{unique}(\mathit{this}, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\})$, can be regained from two $\mathsf{half}(\mathit{this}, \mathsf{open})$ permissions. half is the only permission with an explicit fraction function. All other specifications implicitly quantify over all fraction functions and leave them unchanged.

State Invariants Map Typestates to Fields. We now have a sufficient specification for both sides of the pipe. In order to verify their implementations we need to know what typestates correspond to in implementations. Our implementation verification extends Fugue's approach of using *state invariants* to map states to predicates that describe the fields of an object in a given state [12]. We leverage our hierarchical state spaces and allow state invariants for states with refinements to capture invariants common to all substates of a state.

Figure 7 shows that the source's state invariants describe its three states in the obvious way based on the field `snk` pointing to the sink. Notice that the invariant does not only talk about the sink's state (as in Fugue) but uses permissions to control access through fields just as through local variables.

The sink's state invariants are much more involved (figure 8) and define, e.g., what the difference between an empty buffer ($in < 0$) and a filled circular buffer ($in = out$) is. Interestingly, these invariants are all meticulously documented in the original Java standard library implementation for `PipedInputStream` [4]. The half permission to itself that the sink temporarily holds for the time between calls to `receivedLast` and `close` lets us verify that `close` is allowed to close the sink.

Verification with Invariants. Implementation checking assumes state invariants implied by incoming permissions and tracks changes to fields. Objects *have to be in a state whenever they yield control to another object*, including during method calls. For example, the source transitions to sending before calling the sink. However, the writer never finds the source in the sending state but always ready—sending never occurs in a method specification. We call states that are not observed by a client *intermediate states*. They help us deal with re-entrant calls (details in section 5.2). A practical syntax could make such intermediate states implicit.

Figures 7 and 8 show how implementation checking proceeds for most of the source's and sink's methods. We show in detail how field assignments change the sink's state. The sink's state information is frequently a disjunction of possible states. Dynamic tests essentially rule out states based on incompatible invariants. *All* of these tests are present in the original Java implementation; we removed additional non-null and state tests that are obviated by our approach. This not only shows how our approach forces necessary state tests but also suggests that our specifications could be used to *generate* such tests automatically.

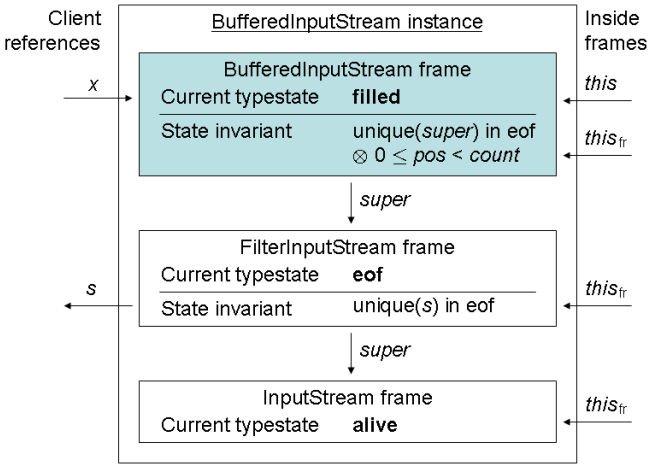

Figure 9. Frames of a `BufferedInputStream` instance in state `filled`. The blue *virtual frame* is in a different state than its super-frame.

3.2 Buffered Input Streams

A `BufferedInputStream` (or "buffer", for short) wraps another "underlying" stream and provides buffering of characters for more efficient retrieval. We will use this example to illustrate our approach to handling inheritance. Compared to the original implementation, we made fields "private" in order to illustrate calls to overridden methods using `super`. We omit intermediate states in this specification.

Class Hierarchy. `BufferedInputStream` is a subclass of `FilterInputStream`, which in turn is a subclass of `InputStream`. `InputStream` is the abstract base class of all input streams and defines their protocol with informal documentation that we formalize in figure 10. It implements *convenience methods* such as `read(int[])` in terms of other—abstract—methods. `FilterInputStream` holds an underlying stream in a field `s` and simply forwards all calls to that stream (figure 10). `BufferedInputStream` overrides these methods to implement buffering.

Frames. The buffer occasionally calls overridden methods to read from the underlying stream. How can we reason about these internal calls? Our approach is based on Fugue's *frames* for reasoning about inheritance [12]. Objects are broken into frames, one for each class in the object's class hierarchy. A frame holds the fields defined in the corresponding class. We call the frame corresponding to the object's runtime type the *virtual frame*, referred to with normal references (including *this*). Relative to a method, we call the current frame—corresponding to the class that the method is defined in—with $\mathit{this}_{\mathsf{fr}}$, and the frame corresponding to

```
public abstract class InputStream {
  states open, closed refine alive;
  states within, eof refine open;

  public abstract int read() :
    share(this_fr, open) ⊸ (result ≥ 0 ⊗ share(this_fr, open))
                         ⊕ (result = −1 ⊗ share(this_fr, open) in eof)
  public abstract void close() :
    full(this_fr, alive) in open ⊸ full(this_fr, alive) in closed

  public int read(byte[] buf) :
    share(this, open) ⊗ buf ≠ null ⊸
      (result = −1 ⊗ share(this, open) in eof) ⊕
      (result ≥ 0 ⊗ share(this, open))
  { ... for(...)
      ... int c = this.read() ... }
}

public class FilterInputStream extends InputStream {
  within := unique(s) in within
  eof := unique(s) in eof
  closed := unique(s) in closed

  private volatile InputStream s;

  protected FilerInputStream(InputStream s)
    unique(s, alive) in open ⊸ unique(this_fr, alive) in open
  { this.s = s; }
  ... // read() and close() forward to s
}
```

Figure 10. Java `FilterInputStream` forwards all calls to underlying `InputStream` (simplified)

the immediate superclass is called *super* frame. Figure 9 shows a sample `BufferedInputStream` instance with its three frames.

Frame Permissions. In our approach, a permission actually grants access to a particular frame. The permissions we have seen so far give a client access to the referenced object's virtual frame. Permissions for other frames are only accessible from inside a subclass through *super*.

Figure 9 illustrates that a `BufferedInputStream`'s state can differ from the state its filter frame is in: the filter's state might be eof (when the underlying stream reaches eof) while the buffer's is still within (because the buffer array still holds unread characters). The state invariants in figure 11 formalize this. They let us verify that `super` calls in the buffer implementation respect the filter's protocol.

Because the states of frames can differ it is important to enforce that a permission is only ever used to access fields in the frame it grants permission to. In specifications we specifically mark permissions that will actually access fields (and not just call other methods) of the receiver with $this_{fr}$. We require all methods that use these permissions to be overridden. On the other hand, convenience methods such as `read(int[])` can operate with permissions to the virtual frame and need not be overridden (figure 10).

```
public class BufferedInputStream
    extends FilterInputStream {
  states depleted, filled refine within;

  closed := unique(super) in closed ⊗ buf = null
  open := unique(buf)
  filled := pos < count ⊗ unique(super) in open
  depleted := pos ≥ count ⊗ unique(super) in within
  eof := pos ≥ count ⊗ unique(super) in  in eof

  private byte buf[] = new byte[8192];
  private int count = 0, pos = 0;

  public BufferedInputStream(InputStream s)
    unique(s) in open ⊸ unique(this_fr) in open
  {                              count = pos = 0 ⊗ unique(buf)
    super(s);                          unique(super) in open
  }                              unique(this_fr, alive) in open

  public synchronized int read() {
    if(pos >= count)
    {                    share(this_fr, open) in depleted ⊕ eof
      fill();               share(this_fr, open) in filled ⊕ eof
      if(pos >= count)
        return -1;          returns share(this_fr, open) in eof
    }                    any path: share(this_fr, open) in filled
    return buf[pos++] & 0xFF;
  }                             share(this_fr, open) in filled ⊕ eof

  private void fill()
    share(this_fr, open) in depleted ⊕ eof ⊸
      share(this_fr, open) in filled ⊕ eof
  {                  invariant: unique(super) in within ⊕ eof
    count = pos = 0;      note: assumes buffer was fully read
    int b = super.read();  unique(super) in within ⊕ eof
    while(b >= 0) {                unique(super) in within
      buf[count++] = (byte) (b & 0xFF);
                                   share(this_fr, open) in filled
      if(count >= buf.length) break;
      b = super.read();      unique(super) in within ⊕ eof
    }             if loop never taken, share(this_fr, open) in eof
  }                            share(this, open) in filled ⊕ eof

  public synchronized void close() {
    buf = null;            invariant: unique(super) in open
    super.close();                unique(super) in closed
  }                             full(this_fr, alive) in closed
```

Figure 11. `BufferedInputStream` caches characters from `FilterInputStream` base class

This distinction implies that `fill` (figure 11) *cannot* call `read(int[])` (because it does not have a suitable virtual frame permission) but *only* `super.read()`. This is imperative for the correctness of `fill` because a dynamically dispatched call would lead back into the—still empty—buffer, causing an infinite loop. (One can trigger exactly this effect in the Java 6 implementation of `BufferedInputStream`.)

3.3 Summary

This section showed how our approach can be used to verify realistic Java pipe and buffered input stream implementations. The notion of access permissions is central to our approach. Overall, we introduced five different kinds of permissions (figure 1). While three kinds are adapted from existing work [7, 12] we recently proposed full and pure permissions [3]. State guarantees and temporary state information increase the usefulness of "weak" (share and pure) permissions. Permission splitting and joining is flexible enough to model temporary aliasing on the stack (during method calls) and in the heap (e.g., in pipes and iterators). Permission-based state invariants enable reasoning about protocol implementations. We handle inheritance based on frames [12] and permit dynamic dispatch within objects for convenience methods.

4. Formal Language

This section formalizes an object-oriented language with protocol specifications. We briefly introduce expression and class declaration syntax before defining state spaces, access permissions, and permission-based specifications. Finally, we discuss handling of inheritance and enforcement of behavioral subtyping.

4.1 Syntax

Figure 12 shows the syntax of a simple class-based object-oriented language. The language is inspired by Featherweight Java (FJ, [24]); we will extend it to include typestate protocols in the following subsections. We identify classes (C), methods (m), and fields (f) with their names. As usual, x ranges over variables including the distinguished variable *this* for the receiver object. We use an overbar notation to abbreviate a list of elements. For example, $\overline{x:T} = x_1{:}T_1,\ \ldots,\ x_n{:}T_n$. Types ($T$) in our system include Booleans (`bool`) and classes.

Programs are defined with a list of class declarations and a main expression. A class declaration CL gives the class a unique name C and defines its fields, methods, typestates, and state invariants. A constructor is implicitly defined with the class's own and inherited fields. Fields (F) are declared with their name and type. Each field is mapped into a part of the state space n that can depend on the field (details in section 5.2). A method (M) declares its result type, formal parameters, specification and a body expression. State refinements R will be explained in the next section; method specifications MS and state invariants N are deferred to section 4.4.

We syntactically distinguish pure terms t and possibly effectful expressions e. Arguments to method calls and object construction are restricted to terms. This simplifies reasoning about effects [30, 9] by making execution order explicit.

Notice that we syntactically restricts field access and assignments to fields of the receiver class. Explicit "getter" and "setter" methods can be defined to give other objects access to fields. Assignments evaluate to the *previous* field value.

programs	PR	::=	$\langle \overline{CL}, e \rangle$
class decl.	CL	::=	$\texttt{class C extends C}' \{ \overline{\texttt{F}}\ \overline{\texttt{R}}\ \texttt{I}\ \overline{\texttt{N}}\ \overline{\texttt{M}} \}$
field decl.	F	::=	$f : T\ \texttt{in}\ n$
meth. decl.	M	::=	$T\ m(\overline{T\ x}) : MS = e$
state decl.	R	::=	$d = \overline{s}\ \texttt{refines}\ s_0$
terms	t	::=	$x \mid o \mid \texttt{true} \mid \texttt{false}$
		$\mid$	$t_1\ \texttt{and}\ t_2 \mid t_1\ \texttt{or}\ t_2 \mid \texttt{not}\ t$
expressions	e	::=	$t \mid f \mid \texttt{assign}\ f := t$
		$\mid$	$\texttt{new}\ C(\overline{t}) \mid t_0.m(\overline{t}) \mid \texttt{super}.m(\overline{t})$
		$\mid$	$\texttt{if}(t, e_1, e_2) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$
values	v	::=	$o \mid \texttt{true} \mid \texttt{false}$
references	r	::=	$x \mid f \mid o$
types	T	::=	$C \mid \texttt{bool}$
nodes	n	::=	$s \mid d$

classes	C	*fields*	f	*variables*	x	*objects*	o
methods	m	*states*	s	*dimensions*	d		

Figure 12. Core language syntax. Specifications (*I, N, MS*) in figure 14.

4.2 State Spaces

State spaces are formally defined as a list of state refinements (see figure 12). A state refinement (R) refines an existing state in a new dimension with a set of mutually exclusive sub-states. We use s and d to range over state and dimension names, respectively. A *node* n in a state space can be a state or dimension. State refinements are inherited by subclasses. We assume a root state alive that is defined in the root class `Object`.

We define a variety of helper judgments for state spaces in figure 13. refinements(C) determines the list of state refinements available in class C. $C \vdash A$ wf defines well–formed state assumptions. Assumptions A combine states and are defined in figure 14. Conjunctive assumptions have to cover orthogonal parts of the state space. $C \vdash n \leq n'$ defines the substate relation for a class. $C \vdash A \# A'$ defines orthogonality of state assumptions. A and A' are orthogonal if they refer to different (orthogonal) state dimensions. $C \vdash A \prec n$ defines that a state assumption A only refers to states underneath a root node n. $C \vdash A \ll n$ finds the tightest such n.

4.3 Access Permissions

Access permissions p give references permission to access an object. Permissions to objects are written access(r, n, g, k, A) (figure 14). (We wrote *perm*(r, n, g) `in` A before.) The additional parameter k allows us to uniformly represent all permissions as explained below.

- Permissions are granted to references r. References can be variables, locations, and fields.
- Permissions apply to a particular *subtree* in the space space of r that is identified by its root node n. It rep-

$$\frac{}{\mathsf{refinements}(\texttt{Object}) = \cdot} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \overline{F}\ \overline{R}\ \ldots\ \}\quad \mathsf{refinements}(C') = \overline{R'}}{\mathsf{refinements}(C) = \overline{R'}, \overline{R}} \qquad \frac{n\ \mathit{in}\ \mathsf{refinements}(C)}{C \vdash n\ \mathsf{wf}}$$

$$\frac{C \vdash A_1\ \mathsf{wf}\quad C \vdash A_2\ \mathsf{wf}}{C \vdash A_1 \oplus A_2\ \mathsf{wf}} \qquad \frac{C \vdash A_1\ \mathsf{wf}\quad A_1\ \#\ A_2\quad C \vdash A_2\ \mathsf{wf}}{C \vdash A_1 \otimes A_2\ \mathsf{wf}} \qquad \frac{d = \overline{s}\ \texttt{refines}\ s \in \mathsf{refinements}(C)}{C \vdash s_i \le d\quad C \vdash d \le s} \qquad \frac{C \vdash n\ \mathsf{wf}}{C \vdash n \le n}$$

$$\frac{C \vdash n \le n''\quad C \vdash n'' \le n'}{C \vdash n \le n'} \qquad \frac{d = \overline{s}\ \texttt{refines}\ s^* \in \mathsf{refinements}(C)\quad d' = \overline{s'}\ \texttt{refines}\ s^* \in \mathsf{refinements}(C)\quad d \ne d'}{C \vdash d\ \#\ d'}$$

$$\frac{C \vdash n_1 \le n_1'\quad C \vdash n_1'\ \#\ n_2'\quad C \vdash n_2 \le n_2'}{C \vdash n_1\ \#\ n_2} \qquad \frac{C \vdash A'\ \#\ A}{C \vdash A\ \#\ A'} \qquad \frac{C \vdash A_{1,2}\ \#\ A}{C \vdash A_1 \otimes A_2\ \#\ A} \qquad \frac{C \vdash A_{1,2}\ \#\ A}{C \vdash A_1 \oplus A_2\ \#\ A} \qquad \frac{C \vdash n' \le n}{C \vdash n' \prec n}$$

$$\frac{C \vdash A_{1,2} \prec n\quad C \vdash A_1 \otimes A_2\ \mathsf{wf}}{C \vdash A_1 \otimes A_2 \prec n} \qquad \frac{C \vdash A_{1,2} \prec n\quad C \vdash A_1 \oplus A_2\ \mathsf{wf}}{C \vdash A_1 \oplus A_2 \prec n} \qquad \frac{C \vdash A \prec n\quad \forall n' : C \vdash A \prec n'\ \mathit{implies}\ n \le n'}{C \vdash A \ll n}$$

Figure 13. State space judgments (assumptions A defined in figure 14)

resents a *state guarantee* (section 3). Other parts of the state space are unaffected by the permission.

- The *fraction function* g tracks for each node on the path from n to alive a symbolic fraction [6]. The fraction function keeps track of how often permissions were split at different nodes in the state space so they can be coalesced later (see section 5.5).
- The *subtree fraction* k encodes the level of access granted by the permission. $k > 0$ grants modifying access. $k < 1$ implies that other potentially modifying permissions exist. Fraction variables z are conservatively treated as a value between 0 and 1, i.e., $0 < z < 1$.
- An *state assumption* A expresses state knowledge within the permission's subtree. Only full permissions can permanently make state assumptions until they modify the object's state themselves. For weak permissions, the state assumption is *temporary*, i.e. lost after any effectful expression (because the object's state may change without the knowledge of r).

We can encode unique, full, share, and pure permissions as follows. In our formal treatment we omit immutable permissions, but it is straightforward to encode them with an additional "bit" that distinguishes immutable and share permissions.

$$\begin{aligned}
\mathsf{unique}(r,n,g)\ \texttt{in}\ A &\equiv \mathsf{access}(r,n,\{g, n \mapsto 1\},1,A)\\
\mathsf{full}(r,n,g)\ \texttt{in}\ A &\equiv \mathsf{access}(r,n,g,1,A)\\
\mathsf{share}(r,n,g,k)\ \texttt{in}\ A &\equiv \mathsf{access}(r,n,g,k,A) \quad (0 < k < 1)\\
\mathsf{pure}(n,n,g)\ \texttt{in}\ A &\equiv \mathsf{access}(r,n,g,0,A)
\end{aligned}$$

4.4 Permission-Based Specifications

We combine atomic permissions (p) and facts about Boolean values (q) using linear logic connectives (figure 14). We also include existential ($\exists z : H.P$) and universal quantification of fractions ($\forall z : H.P$) to alleviate programmers from writing concrete fraction functions in most cases. We type all expressions as an existential type (E).

permissions	p	::=	$\mathsf{access}(r,n,g,k,A)$
facts	q	::=	$t = \texttt{true} \mid t = \texttt{false}$
assumptions	A	::=	$n \mid A_1 \otimes A_2 \mid A_1 \oplus A_2$
fraction fct.	g	::=	$z \mid \overline{n \mapsto v}$
		$\mid$	$g/2 \mid g_1, g_2$
fractions	k	::=	$1 \mid 0 \mid z \mid k/2$
predicates	P	::=	$p \mid q$
		$\mid$	$P_1 \otimes P_2 \mid \mathbf{1}$
		$\mid$	$P_1\ \&\ P_2 \mid \top$
		$\mid$	$P_1 \oplus P_2 \mid \mathbf{0}$
		$\mid$	$\exists z : H.P \mid \forall z : H.P$
method specs	MS	::=	$P \multimap E$
expr. types	E	::=	$\exists x : T.P$
state inv.	N	::=	$n = P$
initial state	I	::=	$\texttt{initially}\ \langle \exists \overline{f : T}.P, S \rangle$
precise state	S	::=	$s_1 \otimes \ldots \otimes s_n$
fract. terms	h	::=	$g \mid k$
fract. types	H	::=	$\mathsf{Fract} \mid \overline{n} \to \mathsf{Fract}$
fract. vars.	z		

Figure 14. Permission-based specifications

Method specifications. Methods are specified with a linear implication ($\multimap$) of predicates (MS). The left-hand side of the implication (method pre-condition) may refer to method receiver and formal parameters. The right-hand side (post-condition) existentially quantifies the method result (a similar technique is used in Vault [11]). We refer to the receiver with *this* and usually call the return value *result*.

State invariants. We decided to use linear logic predicates for state invariants as well (N). In general, several of the defined state invariants will have to be satisfied at the same time. This is due to our hierarchical state spaces. Each class declares an initialization predicate and a start state (I) that are used for object construction (instead of an explicit constructor).

4.5 Handling Inheritance

Permissions give access to a particular frame, usually the virtual frame (see section 3.2) of an object. Permissions to the virtual frame are called *object permissions*. Because of subtyping, the precise frame referenced by an object permission is statically unknown.

$$\textit{references} \quad r \quad ::= \quad \ldots \mid \texttt{super} \mid \texttt{this}_{\mathsf{fr}}$$

In order to handle inheritance, we distinguish references to the receiver's "current" frame (*this*$_{\mathsf{fr}}$) and its super-frame (*super*). Permissions for these "special" references are called *frame permissions*. A *this*$_{\mathsf{fr}}$ permission grants access to fields and can be used in method specifications. Permissions for `super` are needed for super-calls and are only available in state invariants. All methods requiring a *this*$_{\mathsf{fr}}$ permission must be overridden because such methods rely on being defined in a particular frame to access its fields.

4.6 Behavioral Subtyping

Subclasses should be allowed to define their own specifications, e.g. to add precision or support additional behavior [4]. However, subclasses need to be *behavioral subtypes* [29] of the extended class. Our system enforces behavioral subtyping in two steps. Firstly, state space inheritance conveniently guarantees that states of subclasses *always* correspond to states defined in superclasses [4]. Secondly, we make sure that every overriding method's specification implies the overridden method's specification [4] using the override judgment (figure 16) that is used in checking method declarations. This check leads to method specifications that are contra-variant in the domain and co-variant in the range as required by behavioral subtyping.

5. Modular Typestate Verification

This section describes a static modular typestate checking technique for access permissions similar to conventional typechecking. It guarantees at compile-time that protocol specifications will never be violated at runtime. We emphasize that our approach does not require tracking typestates at run time.

A companion technical report contains additional judgments and a soundness proof for a fragment of the system presented in this paper [5]. The fragment does not include inheritance and only supports permissions for objects as a whole. State dimensions are omitted and specifications are deterministic. The fragment does include full, share, and pure permissions with fractions and temporary state information.

5.1 Permission Tracking

We permission-check an expression e with the judgment $\Gamma; \Delta \vdash^{i}_{C} e : \exists x : T.P \setminus \mathcal{E}$. This is read as, "in valid context Γ and linear context Δ, an expression e executed within receiver class C has type T, yields permissions P, and affects fields $\mathcal{E}$". Permissions Δ are consumed in the process. We omit the receiver C where it is not required for checking a particular syntactic form. The set $\mathcal{E}$ keeps track of fields that were assigned to, which is important for the correct handling of permissions to fields. It is omitted when empty. The marker i in the judgment can be 0 or 1 where $i = 1$ indicates that states of objects in the context may change during evaluation of the expression. This will help us reason about temporary state assumptions. A combination of markers with $i \vee j$ is 1 if at least one of the markers is 1.

$$\begin{array}{rrcl} \textit{valid contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : T \mid \Gamma, z : H \mid \Gamma, q \\ \textit{linear contexts} & \Delta & ::= & \cdot \mid \Delta, P \\ \textit{effects} & \mathcal{E} & ::= & \cdot \mid \mathcal{E}, f \end{array}$$

Valid and linear contexts distinguish valid (permanent) information (Γ) from resources (Δ). Resources are tracked linearly, forbidding their duplication, while facts can be used arbitrarily often. (In logical terms, contraction is defined for facts only). The valid context types object variables, fraction variables, and location types and keeps track of facts about terms q. Fraction variables are tracked in order to handle fraction quantification correctly. The linear context holds currently available resource predicates.

The judgment $\Gamma \vdash t : T$ types terms. It includes the usual rule for subsumption based on nominal subtyping induced by the extends relation (figure 16). Term typing is completely standard and can be found in the companion report. The companion report also includes rules for formally typing fractions and fraction functions [5].

Our expression checking rules are syntax-directed up to reasoning about permissions. Permission reasoning is deferred to a separate judgment $\Gamma; \Delta \vdash P$ that uses the rules of linear logic to prove the availability of permissions P in a given context. This judgment will be discussed in section 5.5. Permission checking rules for most expressions appear in figure 15 and are described in turn. Packing, method calls, and field assignment are discussed in following subsections. Helper judgments are summarized in figure 16. The notation $[e'/x]e$ substitutes e' for occurrences of x in e.

- P-TERM embeds terms. It formalizes the standard logical judgment for existential introduction and has no effect on existing objects.
- P-FIELD checks field accesses analogously.
- P-NEW checks object construction. The parameters passed to the constructor have to satisfy initialization predicate P and become the object's initial field values. The new existentially quantified object is associated with a unique permission to the root state that makes state assumptions according to the declared start state A. Object construction has no effect on existing objects.

  The judgment init (figure 16) looks up initialization predicate and start state for a class. The start state is a conjunction of states (figure 14). The initialization predicate is the invariant needed for the start state.
- P-IF introduces non-determinism into the system, reflected by the disjunction in its type. We make sure that the predicate is of Boolean type and then assume its truth

$$\frac{\Gamma \vdash t : T \quad \Gamma; \Delta \vdash [t/x]P}{\Gamma; \Delta \vdash^0 t : \exists x : T.P} \text{ P-TERM} \qquad \frac{\mathsf{localFields}(C) = \overline{f : T} \quad \Gamma; \Delta \vdash [f_i/x]P}{\Gamma; \Delta \vdash^0_C f_i : \exists x : T_i.P} \text{ P-FIELD}$$

$$\frac{\Gamma \vdash \overline{t : T} \quad \mathsf{init}(C) = \langle \exists \overline{f : T}.P, A \rangle \quad \Gamma; \Delta \vdash [\overline{t/f}]P}{\Gamma; \Delta \vdash^0 \mathtt{new}\ C(\overline{t}) : \exists x : C.\mathsf{access}(x, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, 1, A)} \text{ P-NEW}$$

$$\frac{\begin{array}{c}(\Gamma, t = \mathtt{true}); \Delta \vdash^i e_1 : \exists x : T.P_1 \setminus \mathcal{E}_1 \\ \Gamma \vdash t : \mathtt{bool} \quad (\Gamma, t = \mathtt{false}); \Delta \vdash^j e_2 : \exists x : T.P_2 \setminus \mathcal{E}_2\end{array}}{\Gamma; \Delta \vdash^{i \vee j} \mathtt{if}(t, e_1, e_2) : \exists x : T.P_1 \oplus P_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2} \text{ P-IF}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash^i e_1 : \exists x : T.P \setminus \mathcal{E}_1 \quad (\Gamma, x : T); (\Delta', P) \vdash^j e_2 : E_2 \setminus \mathcal{E}_2 \\ i = 1 \textit{ implies no temporary assumptions in } \Delta' \quad \textit{Fields in } \mathcal{E}_1 \textit{ do not occur in } \Delta'\end{array}}{\Gamma; (\Delta, \Delta') \vdash^{i \vee j} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : E_2 \setminus \mathcal{E}_1 \cup \mathcal{E}_2} \text{ P-LET}$$

$$\frac{(\overline{x : T}, \mathtt{this} : C); P \vdash^i_C e : \exists \textit{result} : T_r.P_r \otimes \top \setminus \mathcal{E} \quad E = \exists \textit{result} : T_r.P_r \quad \mathsf{override}(m, C, \forall \overline{x : T}.P \multimap E)}{T_r\ m(\overline{T\ x}) : P \multimap E = e\ \mathsf{ok\ in}\ C} \text{ P-METH}$$

$$\frac{\ldots \quad \overline{M}\ \mathsf{ok\ in}\ C \quad \overline{M}\ \textit{overrides all methods with this}_{\mathit{fr}}\ \textit{permissions in } C'}{\mathtt{class\ C\ extends\ C'}\ \{\ \overline{\mathtt{F}}\ \overline{\mathtt{R}}\ \mathtt{I}\ \overline{\mathtt{N}}\ \overline{\mathtt{M}}\ \}\ \mathsf{ok}} \text{ P-CLASS} \qquad \frac{\overline{CL}\ \mathsf{ok} \quad \cdot; \cdot \vdash^i_{\_} e : E \setminus \mathcal{E}}{\langle \overline{CL}, e \rangle : E} \text{ P-PROG}$$

Figure 15. Permission checking for expressions (part 1) and declarations

$$\frac{\mathtt{class}\ C\ \mathtt{extends}\ C'\ \{\ldots\} \in \overline{CL}}{C\ \mathsf{extends}\ C'} \qquad \frac{\mathtt{class}\ C\ \{\ldots\ \overline{M}\ \ldots\} \in \overline{CL} \quad T_r\ m(\overline{T\ x}) : P \multimap \exists \textit{result} : T_r.P' = e \in \overline{M}}{\mathsf{mtype}(m, C) = \forall \overline{x : T}.P \multimap \exists \textit{result} : T_r.P'}$$

$$\frac{C\ \mathsf{extends}\ C' \quad \mathsf{mtype}(m, C') = \forall \overline{x : T}.MS' \textit{ implies } (\overline{x : T}, \mathtt{this} : C); \cdot \vdash MS \multimap MS'}{\mathsf{override}(m, C, \forall \overline{x : T}.MS)} \qquad \frac{\mathtt{class}\ C \ldots\ \{\overline{F}\ \ldots\} \in \overline{CL}}{\mathsf{localFields}(C) = \overline{F}}$$

$$\frac{}{\mathsf{init}(\mathtt{Object}) = (\mathbf{1}, \mathsf{alive})} \qquad \frac{\begin{array}{c}\mathtt{class}\ C\ \mathtt{extends}\ C'\ \{\overline{f : T\ \mathtt{in}\ n}\ \overline{S}\ \mathtt{initially}\ \langle \exists \overline{f' : T'}, \overline{f : T}.P' \otimes P, A \rangle\ \ldots\} \\ \mathsf{init}(C') = (\exists \overline{f' : T'}.P', A') \quad \cdot; (P, \mathsf{full}(\mathtt{super}, \mathsf{alive}, \{\mathsf{alive} \mapsto 1\}, A')) \vdash \mathsf{inv}_C(A) \otimes \top\end{array}}{\mathsf{init}(C) = \langle \exists \overline{f' : T'}, \overline{f : T}.P' \otimes P, A \rangle}$$

$$\frac{\mathtt{class}\ C\ \{\ldots\ n = P \ldots\} \in \overline{CL}}{\mathsf{pred}_C(n) = P} \qquad \frac{P = \bigotimes_{n' \leq n'' < n} \mathsf{pred}_C(n'')}{\mathsf{pred}_C(n', n) = P} \qquad \frac{\mathsf{inv}_C(A) = P \Rightarrow n'}{\mathsf{inv}_C(n, A) = P \otimes \mathsf{pred}_C(n', n) \otimes \mathsf{pred}_C(n)}$$

$$\frac{}{\mathsf{inv}_C(n) = \mathbf{1} \Rightarrow n} \qquad \frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \quad \mathsf{pred}_C(n_i, n) = P'_i \quad n_1 \otimes n_2 \ll n \quad (i = 1, 2)}{\mathsf{inv}_C(A_1 \otimes A_2) = P_1 \otimes P'_1 \otimes P_2 \otimes P'_2 \Rightarrow n}$$

$$\frac{\mathsf{inv}_C(A_i) = P_i \Rightarrow n_i \quad \mathsf{pred}_C(n_i, n) = P'_i \quad n_1 \oplus n_2 \ll n \quad (i \in 1, 2)}{\mathsf{inv}_C(A_1 \oplus A_2) = (P_1 \otimes P'_1) \oplus (P_2 \otimes \mathsf{pred}_C(n_2, n)) \Rightarrow n}$$

$$\frac{\textit{only pure permissions in } P}{\mathsf{effectsAllowed}(P) = 0} \qquad \frac{\textit{exists share or full permission in } P}{\mathsf{effectsAllowed}(P) = 1}$$

Figure 16. Protocol verification helper judgments

(falsehood) in checking the *then* (*else*) branch. This approach lets branches make use of the tested condition.

- P-LET checks a `let` binding. The linear context used in checking the second subexpression must not mention fields affected by the first expression. This makes sure that outdated field permissions do not "survive" assignments or packing. Moreover, temporary state information is dropped if the first subexpression has side effects.

A program consists of a list of classes and a main expression (P-PROG, figure 15). As usual, the class table $\overline{CL}$ is globally available. The main expression is checked with initially empty contexts. The judgment CL ok (P-CLASS) checks a class declaration. It checks fields, states, and invariants for syntactic correctness (omitted here) and verifies consistency between method specifications and implementations using the judgment M ok in C. P-METH assumes the specified pre-condition of a method (i.e. the left-hand side of the linear implication) and verifies that the method's body expression produces the declared post-condition (i.e. the right-hand side of the implication). Conjunction with $\top$ drops excess permissions, e.g., to dead objects. The override judgment concisely enforces behavioral subtyping (see section 4.6). A method itself is not a linear resource since all resources it uses (including the receiver) are passed in upon invocation.

5.2 Packing and Unpacking

We use a refined notion of *unpacking* [12] to gain access to fields: we unpack and pack a specific permission. The access we gain reflects the permission we unpacked. Full and shared permissions give modifying access, while a pure permission gives read-only access to underlying fields.

To avoid inconsistencies, objects are always fully packed when methods are called. To simplify the situation, only one permission can be unpacked at the same time. Intuitively, we "focus" [13] on that permission. This lets us unpack share like full permissions, gaining full rather than shared access to underlying fields (if available). The syntax for packing and unpacking is as follows.

$$\begin{array}{lrcll} \textit{expressions} & e & ::= & \ldots & \mid \ \texttt{unpack}(n,k,A)\ \texttt{in}\ e \\ & & & & \mid \ \texttt{pack to}\ A\ \texttt{in}\ e \end{array}$$

Packing and unpacking always affects the receiver of the currently executed method. The `unpack` parameters express the programmer's expectations about the permission being unpacked. For simplicity, an explicit subtree fraction k is part of `unpack` expressions. It could be inferred from a programmer-provided permission kind, e.g. share.

Typechecking. In order for `pack` to work properly we have to "remember" the permission we unpacked. Therefore we introduce unpacked as an additional linear predicate.

$$\textit{permissions} \quad p \quad ::= \quad \ldots \mid \mathsf{unpacked}(n,g,k,A)$$

The checking rules for packing and unpacking are given in figure 18. Notice that packing and unpacking always affects permissions to $\texttt{this}_{\mathsf{fr}}$. (We ignore substitution of `this` with an object location at runtime here.)

P-UNPACK first derives the permission to be unpacked. The judgment inv determines a predicate for the receiver's fields based on the permission being unpacked. It is used when checking the body expression. An unpacked predicate is added into the linear context. We can prevent multiple permissions from being unpacked at the same time using a straightforward dataflow analysis (omitted here).

$$\begin{array}{rcl} \mathsf{inv}_C(n,g,k,A) & = & \mathsf{inv}_C(n,A) \otimes \mathsf{purify}(\mathsf{above}_C(n)) \\ \mathsf{inv}_C(n,g,0,A) & = & \mathsf{purify}\,(\mathsf{inv}_C(n,A) \otimes \mathsf{above}_C(n)) \\ \textit{where}\ \mathsf{above}_C(n) & = & \bigotimes_{n':n<n'\leq\mathsf{alive}} \mathsf{pred}_C(n') \end{array}$$

Figure 17. Invariant construction (purify in figure 19)

P-PACK does the opposite of P-UNPACK. It derives the predicate necessary for packing the unpacked permission and then assumes that permission in checking the body expression. The new state assumption A can differ from before only if a modifying permission was unpacked. Finally, the rule ensures that permissions to fields do not "survive" packing.

Invariant transformation. The judgment $\mathsf{inv}_C(n,g,k,A)$ determines what permissions to fields are implied by a permission $\mathsf{access}(\textit{this}_{\mathsf{fr}},n,g,k,A)$ for a frame of class C. It is defined in figure 17 and uses a purify function (figure 19) to convert arbitrary into pure permissions.

Unpacking a full or shared permission with root node n yields purified permissions for nodes "above" n and includes invariants following from state assumptions as–is. Conversely, unpacking a pure permission yields completely purified permissions.

5.3 Calling Methods

Checking a method call involves proving that the method's pre-condition is satisfied. The call can then be typed with the method's post-condition.

Unfortunately, calling a method can result into reentrant callbacks. In order to ensure that objects are consistent when called we require them to be fully packed before method calls. This reflects that aliased objects always have to be prepared for reentrant callbacks.

This rule is not a limitation because we can always pack to some intermediate state although it may be inconvenient in practice. Notice that such *intermediate packing* obviates the need for adoption while allowing focus [13]: the intermediate state represents the situation where an adopted object was taken out of the adopting object. Inferring intermediate states as well as identifying where reentrant calls are impossible (intermediate packing avoidance) are important areas for future research.

Virtual calls. Virtual calls are dynamically dispatched (rule P-CALL). In virtual calls, frame and object permissions are identical because object permissions simply refer to the object's virtual frame. This is achieved by substituting the given receiver for both *this* and $\textit{this}_{\mathsf{fr}}$.

Super calls. Super calls are statically dispatched (rule P-SUPER). Recall that `super` is used to identify permissions to the super-frame. We substitute `super` only for $\textit{this}_{\mathsf{fr}}$. We omit a substitution of *this* for the receiver (*this* again) for clarity.

$$\frac{\Gamma;\Delta \vdash_C \mathsf{access}(\mathit{this}_{\mathsf{fr}}, n, g, k, A) \quad \textit{receiver packed} \qquad k = 0 \textit{ implies } i = 0 \quad \Gamma;(\Delta', \mathsf{inv}_C(n,g,k,A), \mathsf{unpacked}(n,g,k,A)) \vdash^i_C e : E \setminus \mathcal{E}}{\Gamma;(\Delta,\Delta') \vdash^i_C \mathtt{unpack}(n,k,A)\ \mathtt{in}\ e : E \setminus \mathcal{E}}\ \text{P-UNPACK}$$

$$\frac{\Gamma;\Delta \vdash_C \mathsf{inv}_C(n,g,k,A) \otimes \mathsf{unpacked}(n,g,k,A') \quad k = 0 \textit{ implies } A = A' \qquad \Gamma;(\Delta', \mathsf{access}(\mathit{this}_{\mathsf{fr}}, n, g, k, A)) \vdash^i_C e : E \setminus \mathcal{E} \quad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \quad \textit{Fields do not occur in } \Delta'}{\Gamma;(\Delta,\Delta') \vdash^i_C \mathtt{pack}\ n\ \mathtt{to}\ A\ \mathtt{in}\ e : E \setminus \overline{f}}\ \text{P-PACK}$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad \Gamma \vdash \overline{t : T} \quad \Gamma;\Delta \vdash [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P \qquad \mathsf{mtype}(m, C_0) = \forall \overline{x : T}.P \multimap E \quad i = \mathsf{effectsAllowed}(P) \quad \textit{receiver packed}}{\Gamma;\Delta \vdash^i t_0.m(\overline{t}) : [t_0/\mathtt{this}][t_0/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}\ \text{P-CALL}$$

$$\frac{\Gamma \vdash \overline{t : T} \quad \Gamma;\Delta \vdash [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]P \quad C\ \mathsf{extends}\ C' \qquad \mathsf{mtype}(m, C') = \forall \overline{x : T}.P \multimap E \quad i = \mathsf{effectsAllowed}(P) \quad \textit{receiver packed}}{\Gamma;\Delta \vdash^i_C \mathtt{super}.m(\overline{t}) : [\mathtt{super}/\mathtt{this}_{\mathsf{fr}}][\overline{t}/\overline{x}]E}\ \text{P-SUPER}$$

$$\frac{\Gamma;\Delta \vdash t : \exists x : T_i.P \quad \Gamma;\Delta' \vdash_C [f_i/x']P' \otimes p \qquad \mathsf{localFields}(C) = \overline{f : T\ \mathtt{in}\ n} \quad n_i \leq n \quad p = \mathsf{unpacked}(n,g,k,A), k \neq 0}{\Gamma;(\Delta,\Delta') \vdash^1_C \mathtt{assign}\ f_i := t : \exists x' : T_i.P' \otimes [f_i/x]P \otimes p \setminus f_i}\ \text{P-ASSIGN}$$

Figure 18. Permission checking for expressions (part 2)

$$\frac{p = \mathsf{access}(r,n,g,k,A)}{\mathsf{purify}(p) = \mathsf{pure}(r,n,g,A)} \qquad \frac{\mathsf{purify}(P_1) = P_1' \quad \mathsf{purify}(P_2) = P_2' \quad \mathtt{op} \in \{\otimes, \&, \oplus\}}{\mathsf{purify}(P_1\ \mathtt{op}\ P_2) = P_1'\ \mathtt{op}\ P_2'}$$

$$\frac{\mathtt{unit} \in \{\mathbf{1}, \top, \mathbf{0}\}}{\mathsf{purify}(\mathtt{unit}) = \mathtt{unit}} \qquad \frac{\mathsf{purify}(P) = P'}{\mathsf{purify}(\exists z : H.P) = \exists z : H.P'} \qquad \frac{\mathsf{purify}(P) = P'}{\mathsf{purify}(\forall z : H.P) = \forall z : H.P'}$$

Figure 19. Permission purification

5.4 Field Assignments

Assignments to fields change the state of the receiver's current frame. We point out that assignments to a field do *not* change states of objects referenced by the field. Therefore reasoning about assignments mostly has to be concerned with preserving invariants of the receiver. The unpacked predicates introduced in section 5.2 help us with this task.

Our intuition is that assignment to a field requires unpacking the surrounding object to the point where all states that refer to the assigned field in their invariants are revealed. Notice that the object does not have to be unpacked completely in this scheme. For simplicity, each field is annotated with the subtree that can depend on it (figure 12). Thus we interpret subtrees as data groups [27].

The rule P-ASSIGN (figure 18) assigns a given object t to a field f_i and returns the old field value as an existential x'. This preserves information about that value. The rule verifies that the new object is of the correct type and that a suitable full or share permission is currently unpacked. By recording an effect on f_i we ensure that information about the old field value cannot "flow around" the assignment (which would be unsound).

5.5 Permission Reasoning with Splitting and Joining

Our permission checking rules rely on proving a predicate P given the current valid and linear resources, written $\Gamma;\Delta \vdash P$. We use standard rules for the decidable multiplicative-additive fragment of linear logic (MALL) with quantifiers that only range over fractions [28]. Following Boyland [7] we introduce a notion of substitution into the logic that allows substituting a set of linear resources with an equivalent one.

$$\frac{\Gamma;\Delta \vdash P' \quad P' \Rrightarrow P}{\Gamma;\Delta \vdash P}\ \text{SUBST}$$

The judgment $P \Rrightarrow P'$ defines legal substitutions. We use substitutions for splitting and joining permissions (figure 20). The symbol $\Lleftarrow\Rrightarrow$ indicates that transformations are allowed in both directions. SYM and ASYM generalize the rules from section 2. Most other rules are used to split permissions for larger subtrees into smaller ones and vice versa. A detailed explanation of these rules can be found in the companion report [5].

Our splitting and joining rules maintain a consistent set of permissions for each object so that no permission can ever violate an assumption another permission makes. Fractions

$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r,n,g,k,A) \Longleftrightarrow \mathsf{access}(r,n,g/2,k/2,A') \otimes \mathsf{access}(r,n,g/2,k/2,A'')}\ \textsc{Sym}$$

$$\frac{A = A' = A'' \textit{ or } (A = A' \textit{ and } A'' = n) \textit{ or } (A = A'' \textit{ and } A' = n)}{\mathsf{access}(r,n,g,k,A) \Longleftrightarrow \mathsf{access}(r,n,g/2,k,A') \otimes \mathsf{pure}(r,n,g/2,A'')}\ \textsc{Asym}$$

$$\frac{\begin{array}{c} n_1 \# n_2 \quad A_1 \prec n_1 \leq n \quad A_2 \prec n_2 \leq n \\ p_i = \mathsf{full}(r,n_i,\{g,\mathsf{nodes}(n_i,n) \mapsto 1\}/2,A_i) \end{array}}{\mathsf{full}(r,n,g,A_1 \otimes A_2) \Rrightarrow p_1 \otimes p_2}\ \textsc{F-Split-}\otimes$$

$$\frac{\begin{array}{c} n_1 \# n_2 \quad A_1 \prec n_1 \leq n \quad A_2 \prec n_2 \leq n \\ p_i = \mathsf{full}(r,n_i,\{g,n \mapsto 1,\mathsf{nodes}(n_i,n) \mapsto 1\}/2,A_i) \end{array}}{p_1 \otimes p_2 \Rrightarrow \mathsf{full}(r,n,\{g,n \mapsto 1\},A_1 \otimes A_2)}\ \textsc{F-Join-}\otimes$$

$$\frac{A_1 \# A_2}{\mathsf{full}(r,n,g,A_1 \oplus A_2) \Longleftrightarrow \mathsf{full}(r,n,g,A_1) \oplus \mathsf{full}(r,n,g,A_2)}\ \textsc{F-}\oplus$$

$$\frac{A \prec n' \leq n}{\mathsf{full}(r,n,g,A) \Rrightarrow \mathsf{full}(r,n',\{g,\mathsf{nodes}(n',n) \mapsto 1\},A)}\ \textsc{F-Down}$$

$$\frac{A \prec n' \leq n}{\mathsf{full}(r,n',\{g,n \mapsto 1,\mathsf{nodes}(n',n) \mapsto 1\},A) \Rrightarrow \mathsf{full}(r,n,\{g,n \mapsto 1\},A)}\ \textsc{F-Up}$$

$$\frac{n' \leq n}{\mathsf{pure}(r,n,\{g,\mathsf{nodes}(n',n) \mapsto \overline{k}\},A) \Rrightarrow \mathsf{pure}(r,n',g,A)}\ \textsc{P-Up}$$

$$\frac{}{\mathsf{access}(r,n,g,k,A) \Rrightarrow \mathsf{access}(r,n,g,k,n)}\ \textsc{Forget}$$

Figure 20. Splitting and joining of access permissions

of all permissions to an object sum up to (at most) 1 for every node in the object's state space.

5.6 Example

To illustrate how verification proceeds, figure 21 shows the `fill` method from `BufferedInputStream` (figure 11) written in our core language. As can be seen we need an intermediate state `reads` and a marker field `reading` that indicate an ongoing call to the underlying stream. We also need an additional state refinement to specify an internal method replacing the `while` loop in the original implementation. (We assume that $\mathit{this}_{\mathsf{fr}}$ permissions can be used for calls to `private` methods.)

Maybe surprisingly, we have to reassign field values after `super.read()` returns. The reason is that when calling `super` we lose temporary state information for *this*. Assignment re-establishes this information and lets us pack properly before calling `doFill` recursively or terminating in the cases of a full buffer or a depleted underlying stream.

It turns out that these re-assignments are *not* just an inconvenience caused by our method but point to a real problem in the Java standard library implementation. We could implement a malicious underlying stream that calls back into the "surrounding" `BufferedInputStream` object. This call changes a field, which causes the buffer's invariant on *count* to permanently break, *later on* resulting in an undocumented array bounds exception when trying to read behind the end of the buffer array.

Because `fill` operates on a share permission our verification approach forces taking into account possible field changes through reentrant calls with other share permissions. (This is precisely what our malicious stream does.) We could avoid field re-assignments by having `read` require a full permission, thereby documenting that reentrant (modifying) calls are not permitted for this method.

6. Related Work

In previous work we proposed more expressive typestate specifications [4] that can be verified with the approach presented in this paper. We also recently proposed full and pure permissions and applied our approach to specifying full Java iterators [3]. Verification of protocol compliance has been studied from many different angles including type systems, abstract interpretation, model checking, and verification of general program behavior. Aliasing is a challenge for all of these approaches.

The system that is closest to our work is Fugue [12], the first modular typestate verification system for object-oriented software. Methods are specified with a deterministic state transition of the receiver and pre-conditions on arguments. Fugue's type system tracks objects as "not aliased" or "maybe aliased". Leveraging research on "alias types" [33] (see below), objects typically remain "not aliased" as long as they are only referenced on the stack. Only "not aliased" objects can change state; once an object becomes "maybe

```
class BufferedInputStream extends FilterInputStream {
states ready, reads refine open; ...
states partial, complete refine filled;

reads := reading; ready := reading = false; ...

private boolean reading; ...

public int read() : ∀k : Fract.... =
unpack(open, k) in
  let r = reading in if(r == false, ... fill() ... )

private bool fill() : ∀k : Fract.
  share(this_fr, open) in depleted ⊕ eof ⊸
    share(this_fr, open) in available ⊕ eof =
unpack(open, k, depleted ⊕ eof) in
  assign count = 0 in assign pos = 0 in
  assign reading = true in
  pack to reads in
    let b = super.read() in
    unpack(open, k, open) in
      let r = reading in assign reading = false in
      assign count = 0 in assign pos = 0 in
      if(r, if(b = -1, pack to eof in false,
              pack to depleted in doFill(b)),
            pack to eof in false)

private bool doFill(int b) : ∀k : Fract.
  share(this_fr, open) in depleted ⊕ partial ⊸
    share(this_fr, open) in partial ⊕ complete =
unpack(open, k, depleted ⊕ partial) in
  let c = count in let buffer = buf in
  assign buffer[c] = b in assign count = c + 1 in
  let l = buffer.length in
  if(c + 1 >= l, pack to complete in true,
    assign reading = true in pack to reads in
      let b = super.read() in unpack(open, k) in
        let r = reading in assign reading = false in
        assign count = c + 1 in assign pos = 0 in
        pack to partial in
          if(r == false || b == -1, true, doFill(b))
```

Figure 21. Fragment of `BufferedInputStream` from figure 11 in core language

aliased" its state is permanently fixed although fields can be assigned to if the object's abstract typestate is preserved.

Our work is greatly inspired by Fugue's abilities. Our approach supports more expressive method specifications based on linear logic [18]. Our verification approach is based on "access permissions" that permit state changes even in the presence of aliases. We extend several ideas from Fugue to work with access permissions including state invariants, packing, and frames. Fugue's specifications are expressible with our system [4]. Fugue's "not aliased" objects can be simulated with unique permissions for alive and "maybe aliased" objects correspond to share permissions with state guarantees. There is no equivalent for state dimensions, temporary state assumptions, full, immutable, and pure permissions, or permissions for object parts in Fugue.

Verification of protocol compliance has also been described as "resource usage analysis" [23]. Protocol specifications have been based on very different concepts including typestates [34, 11, 25], type qualifiers [16], size properties [9], direct constraints on ordering [23, 35], and type refinements [30, 10]. None of the above systems can verify implementations of object-oriented protocols like our approach and only two [35, 10] target object-oriented languages. Effective type refinements [30] employ linear logic reasoning but cannot reason about protocol implementations and do not support aliasing abstractions. Hob [25] verifies data structure implementations for a procedural language with static module instantiation based on typestate-like constraints using shape analyses. In Hob, data can have states, but modules themselves cannot. In contrast, we can verify the implementation of stateful objects that are dynamically allocated and support aliasing with permissions instead of shape analysis. Finally, concurrent work on Java(X) proposes "activity annotations" that are comparable to full, share, and pure permissions for whole objects that can be split but not joined. Similar to effective type refinements, state changes can be tracked for a pre-defined set of types, but reasoning about the implementation of these types is not supported. To our knowledge, none of the above systems supports temporary state information.

Because programming with linear types [36] is very inconvenient, a variety of relaxing mechanisms were proposed. Uniqueness, sharing, and immutability (sometimes called read-only) [7] have recently been put to use in resource usage analysis [23, 9]. Alias types [33] allow multiple variables to refer to the same object but require a linear token for object accesses that can be borrowed [7] during function calls. Focusing can be used for temporary state changes of shared objects [13, 16, 2]. Adoption prevents sharing from leaking through entire object graphs (as in Fugue [12]) and allows temporary sharing until a linear adopter is deallocated [13]. All these techniques need to be aware of all references to an object in order to change its state.

Access permissions allow state changes even if objects are aliased from unknown places. Moreover, access permissions give fine-grained access to individual data groups [27]. States and fractions [6] let us capture alias types, borrowing, adoption, and focus with a single mechanism. Sharing of individual data groups has been proposed before [7], but it has not been exploited for reasoning about object behavior. In Boyland's work [6], a fractional permission means immutability (instead of sharing) in order to ensure noninterference of permissions. We use permissions to keep state assumptions consistent but track, split, and join permissions in the same way as Boyland.

Global approaches are very flexible in handling aliasing. Approaches based on abstract interpretation (e.g. [1, 19, 14]) typically verify client conformance while the protocol implementation is assumed correct. Sound approaches rely on a global aliasing analysis [1, 14]. Likewise, most

model checkers operate globally (e.g. [21]) or use assume-guarantee reasoning between coarse-grained static components [17, 22]. The Magic tool checks individual C functions but has to inline user-provided state machine abstractions for library code in order to accommodate aliasing [8]. The above analyses typically run on the complete code base once a system is fully implemented and are very expensive. Our approach supports developers by checking the code at hand like a typechecker. Thus the benefits of our approach differ significantly from global analyses.

Recently, there has been progress in inferring typestate protocols in the presence of aliasing [31], which we believe could be fruitfully combined with our work to reduce initial annotation burden.

Finally, general approaches to specifying program behavior [26, 15, 2] can be used to reason about protocols. The JML [26] is very rich and complex in its specification features; it is more capable than our system to express object behavior (not just protocols), but also potentially more difficult to use due to its complexity. Verifying JML specifications is undecidable in the general case. Tools like ESC/Java [15] can partially check JML specifications but are unsound because they do not have a sound methodology for handling aliasing. Spec# is comparable in its complexity to the JML and imposes similar overhead. The Boogie methodology allows sound verification of Spec# specifications but requires programs to follow an ownership discipline [2].

Our system is much simpler than these approaches, focusing as it does on protocols, and it is designed to be decidable. Our treatment of aliasing makes our system sound, where ESC/Java is not. While the treatment of aliasing in our system does involve complexity, it gives the programmer more flexibility than Boogie's while remaining modular and sound. Because it is designed for protocol verification in particular, our system will generally impose smaller specification overhead than the JML or Spec#.

7. Conclusions

This paper proposes a sound modular protocol checking approach, based on typestates, that allows a great deal of flexibility in aliasing. A novel abstraction, access permissions, combines typestate and object aliasing information. Developers express their protocol design intent using access permissions. Our checking approach then tracks permissions through method implementations. For each object reference the checker keeps track of the degree of possible aliasing and is appropriately conservative in reasoning about that reference. A way of breaking an invariant in a frequently used Java standard library class was exposed in this way. The checking approach handles inheritance in a novel way, giving subclasses more flexibility in method overriding. Case studies on Java iterators and streams provide evidence that access permissions can model realistic protocols, and protocol checking based on access permissions can be used to reason precisely about protocols arising in practice.

In future work we hope to further refine and evaluate our approach. We plan to develop a deterministic algorithm for reasoning about permissions. We hope to leverage our experiences in using our approach to increase its practicality. Based on the case studies presented in this paper we made the following observations:

- In this paper we chose to make the linear logic formalism underlying our approach explicit in example protocol specifications. However, our case studies suggest that practical protocols follow certain patterns. For example, method specifications often consist of simple conjunctions that can be expressed by annotating each method argument separately. With syntactic sugar for such patterns we believe that programmers will only rarely have to use linear logic operators explicitly.
- Specification effort lies primarily with protocol *implementation* developers, which better amortizes over time. Conversely, iterator, stream, and other libraries' clients have (we believe) minimal work to do unless they store objects in fields. (Fugue's experience suggests that loop invariants for typestate checking can often be inferred [12].)
- Only a fraction of our system's capabilities are needed for any given example (although they all are necessary in different situations). Developers do have to understand the general idea of access permissions.

We believe that these observations indicate that the approach can be practical, especially with the help of syntax that captures common cases concisely. A systematic evaluation of this claim is an important part of planned future work.

Acknowledgments

We thank John Boyland, Frank Pfenning, the Plaid group, Sebastian Boßung, and Jason Reed for fruitful discussions on this topic. We also thank the anonymous reviewers for their helpful feedback. This work was supported in part by NASA cooperative agreement NNA05CS30A, NSF grant CCF-0546550, the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", and the U.S. Department of Defense.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of the Eighth 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] K. Bierhoff. Iterator specification with typestates. In *5th Int. Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006.

[4] 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. ACM Press, Sept. 2005.

[5] K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. Technical Report CMU-ISRI-07-105, Carnegie Mellon University, Mar. 2007. http://reports-archive.adm.cs.cmu.edu/anon/isri2007/CMU-ISRI-07-105.pdf.

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

[7] 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.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Int. Conference on Software Engineering*, pages 385–395, May 2003.

[9] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *Int. Conference on Software Engineering*, pages 186–195, May 2005.

[10] M. Degen, P. Thiemann, and S. Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*. Springer, Aug. 2007.

[11] 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.

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

[13] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.

[14] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ACM Int. Symposium on Software Testing and Analysis*, pages 133–144, July 2006.

[15] 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.

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

[17] D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Int. Conference on Software Engineering*, pages 211–220, May 2004.

[18] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[19] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[20] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[22] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *ACM Int. Symposium on Software Testing and Analysis*, pages 39–49. ACM Press, July 2007.

[23] A. Igarashi and N. Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages*, pages 331–342, Jan. 2002.

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999.

[25] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), Dec. 2006.

[26] 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.

[27] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998.

[28] P. Lincoln and A. Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994.

[29] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[30] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ACM Int. Conference on Functional Programming*, pages 213–225, 2003.

[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, 2005.

[32] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM Conference on Programming Language Design and Implementation*, pages 83–94, 2002.

[33] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381. Springer, 2000.

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

[35] G. Tan, X. Ou, and D. Walker. Enforcing resource usage protocols via scoped methods. In *Int. Workshop on Foundations of Object-Oriented Languages*, 2003.

[36] P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.