Typestate-Oriented Programming

Jonathan Aldrich Joshua Sunshine Darpan Saini Zachary Sparks

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
{aldrich,jssunshi,dsaini,zsparks}@andrew.cmu.edu

Abstract

Objects model the world, and state is fundamental to a faithful modeling. Engineers use state machines to understand and reason about state transitions, but programming languages provide little support for reasoning about or implementing these state machines, causing software defects and lost productivity when objects are misused.

We propose *Typestate-Oriented Programming* as a natural extension to the object paradigm, where objects are modeled not just in terms of classes, but in terms of changing *states*. Each state may have its own representation and methods which may transition the object into a new state. A flow-sensitive, permission-based type system helps developers track which state objects are in. First-class typestates are a powerful abstraction that will help developers model and reuse objects more efficiently and correctly.

Categories and Subject Descriptors D.3.1 [*Programming Techniques*]: Miscellaneous—Typestate-oriented programming; D.3.2 [*Programming Languages*]: Language Classifications—Typestate-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—States

General Terms Design, Documentation, Human Factors, Languages, Theory, Verification

1. Introduction

Object-oriented programming was originally developed to facilitate simulation [Dahl and Nygaard 1966], but the object and class abstractions used proved to be a revolutionary advance for effectively building programs in a wide variety of domains. Part of the reason for objects' success is that they are useful not just for simulation of real-world entities, but for building reusable libraries of abstract entities as diverse as collections, streams, and windows.

In the last decade, sophisticated object-oriented libraries and frameworks have leveraged the power of objects to support widespread component-based reuse. While this reuse has enhanced productivity, it has also brought its own problems. Library and framework APIs must often be complex in order to provide rich functionality that can be reused in very general ways. In order to effectively leverage these APIs, programmers must understand how to use them correctly.

One critical aspect of correct API use is *typestate*—an abstraction of the operations currently available on an object, which may change as the program executes [Strom and Yemini 1986]. A familiar example is files that may be open or closed. In the open state, one may read or write to a file, or one may close it, which causes a *state transition* to the closed state. In the closed state, the only permitted operation is to (re-)open the file.

States are a fundamental abstraction in computer science, in engineering more broadly, and in the natural world. Looking just at the Java standard libraries, we see a rich variety of states: streams may be open or closed, iterators may have elements available or not, collections may be empty or not, and even lowly exceptions can have their cause set, or not[1]. `ResultSets` from the JDBC library have dozens of states dealing with different combinations of openness, direction, random access, insertions, etc. Programmers follow engineers and scientists in other disciplines in drawing state diagrams, which show the possible states in a system and transitions between them. Even the natural world has states: are you hungry right now, or full? Perhaps a meal would produce a satisfying state transition.

States influence behavior, and trying to perform an operation on an object in the wrong state is likely to have unfortunate results—as when a code review goes over into lunchtime. While trying to write to a closed file may not be common, it is easy to make state-related mistakes in libraries with less familiar and more complex state spaces. A recent example we observed comes from the ASP.NET framework.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

[1] the cause of an exception can only be set once

In the drop-down list widget, one must de-select one item before selecting another; simply selecting another item will result in an unhelpful exception thrown, with a stack trace that is completely unrelated to the code containing the error. These problems occur, in part, because states are an *implicit* part of component interfaces—they cannot be expressed in language-level interfaces, yet they are essential for using components correctly.

In this paper, we propose *Typestate-Oriented Programming* as a natural extension to the object paradigm, where objects are modeled not just in terms of classes, but in terms of their changing states. In our model, each object state may have its own representation and methods which may transition the object into a new state. A flow-sensitive, permission-based type system helps developers track which state objects are in.

Because states are a natural way of modeling the world, we believe typestate-oriented programming will yield abstraction benefits similar to but beyond those provided by object-oriented programming. Because states are ubiquitous in today's software designs, we believe that explicit language support for typestate will express those designs more clearly than the state of the art. Because the cost of misunderstanding and misusing states is so high, we believe that the improved documentation and checking provided by typestate will prevent errors and enhance developers' productivity. For all these reasons, typestate-oriented programming has the potential to spark a new revolution in component-based software reuse.

2. Context and Approach

Before fleshing out the typestate-oriented programming paradigm in section 3, we consider the historical context and alternative approaches to the problems described in the introduction.

State-Based Designs. Today's programming languages have no direct support for typestate. Instead, the behavioral aspects of state must be encoded via the State design pattern [Gamma et al. 1995], `if`-tests on flag fields, and other indirect mechanisms. While the State pattern offers extensibility, it does not help with larger coordination challenges such as ensuring that a client uses only operations that are available in the current state.

Typestate Checkers. Recent advances in typestate focus on specifying the interface of a class in terms of states and ensuring that clients only call functions that are appropriate in a given state. The Fugue system [Deline and Fahndrich 2004] was the first modular typestate verification system for object-oriented software. The key verification ideas in Fugue include classifying references as not aliased or possibly aliased, associating each class state with an invariant describing the state of the class's fields (which is necessary to support modular verification), and a frame-based approach to supporting verification in the presence of inheritance.

Our Plural system extended Fugue in a number of pragmatically important ways, providing a richer set of aliasing abstractions, a state refinement mechanism that ensures subtyping, a way to make guarantees about the state of shared mutable objects, and a way to dynamically test the state of objects. In a recent case study, we showed that Plural can verify typestate properties such as correct iterator use across a 30,000-line program, as well as verify compliance to protocols in a layered system that aliases stateful objects in interesting ways [Bierhoff et al. 2009].

An alternative to tracking typestate through an enhanced type system is specifying state -achine constraints on how an abstraction is used, then applying static analysis to check whether they are followed [Fink et al. 2008, Bodden et al. 2008]. This pushbutton approach has the advantage of low overhead—the user simply specifies the property, runs the tool and gets messages regarding potential errors. However, the analysis requires good aliasing information to be successful, which is notoriously challenging.

Benefits of a Language-Based Approach. While typestate checking systems are approaching practicality for Java-like languages, there are numerous advantages to building typestate into the object model of the language rather than running a separate checker afterward.

First, language influences thought [Boroditsky 2009]: by incorporating typestate explicitly into a programming language, we encourage library writers and users to think in terms of states, which should help them to design, document, and reuse library components more effectively. Interfaces have states, whether the language does or not; by adding them to the language, we encourage library designers and users to think about them and to write better code.

Second, support for typestate currently faces a barrier of complexity: Java's generics are rightly viewed as complex[2], and typestate adds additional complexity on top of that. Furthermore, mechanisms like type parameterization need to be duplicated for typestate, so that we can talk not only about a list of files, but also about a list of *open* files. By adding typestate directly to the programming language, we can integrate it better and reuse mechanisms between the type system and the state-tracking system, yielding a simpler and more understandable design overall. For example, our proposed design has a single parameterization construct which applies both to traditional types and to states.

Third, putting typestate in the programming language offers new opportunities for expressiveness. An open file has a field holding the operating system file handle; a closed file does not. However, in Java, that field will be present in both the open and closed states, and so the field must be assigned a sentinel value (e.g., `null`) in the closed state.

[2] Wildcards—need we say more?

This makes reasoning more complex, as the maintainer of the class must be aware of a class invariant stating that the field has a sentinel value if and only if the object is in the closed state. If states are supported in the language, we can simply eliminate the field in the closed state. There is never any possibility of misinterpreting the sentinel value as a real one (null pointer exceptions/core dumps anyone?) and reasoning about the invariant is simpler because it is built into the state-based model of the system.

Prior Language-Based Approaches. The Actor model [Hewitt et al. 1973] was one of the first programming models to treat states in a first class way. An Actor can accept one of several messages; in response, it can perform computation, create other actors, send messages, and finally determine its next state—i.e. how to respond to the next message received. Our state-orientated approach draws inspiration from actors, but our concurrency approach [Stork et al. 2009] stays within a call-return function model rather than using messages.

Smalltalk [Kay 1993] introduced a `become` method that allows an object to exchange state and behavior with another object, which can be used to model state changes in a first-class way. In a related approach, the Self language [Ungar and Smith 1987] allows an object to change the objects it delegates to (i.e. inherits from), also providing a way to model state changes.

The concept of a state is related to that of a *role* played in interactions with other object. While most research in the area uses roles to describe different (simultaneous) views of an object, Pernici proposed state-like roles where objects can transition from one role to another [Pernici 1990].

From the object modeling point of view, the closest work to ours is Taivalsaari's proposal to extend class-based languages with explicit definitions of logical states (modes), each with its own set of operations and corresponding implementations [Taivalsaari 1993]. Our proposed object model differs in providing explicit state transitions (rather than implicit ones determined by fields) and in allowing different fields in different states.

A number of CAD tools such as iLogic Rhapsody or IBM/Rational Rose Real-Time support a programming model based on Statecharts [Harel 1987]; such models benefit from many rich features of Statecharts but lack the dynamism of object-oriented systems. Recently Sterkin proposed embedding the principal features of Statecharts as a library within Groovy, providing a smoother integration with objects [Sterkin 2008]. Our approach focuses on adding states to the object-oriented paradigm, and does not consider other features of Statecharts.

Prior Type System Support. Our proposal differs from all the approaches above by providing a type system for tracking state changes. A related notion is allowing class changes in a statically typed language [Bejleri et al. 2006, Bettini et al. 2009]. If changing classes are viewed as states, this is very similar in spirit to our proposal; in fact the Fickle system [Drossopoulou et al. 2001] distinguishes "state classes," which describe states that can change. The recent advances in typestate systems described above have overcome a number of the limitations in Fickle and related systems–e.g. the inability to track the states of fields–and leveraging these advances opens the way to the new paradigm and accompanying language design we describe in this paper.

```
state File {
  public final String filename;
}

state OpenFile extends File {
  private CFilePtr filePtr;
  public int read() { ... }
  public void close() [OpenFile>>ClosedFile]
    { ... }
}

state ClosedFile extends File {
  public void open() [ClosedFile>>OpenFile]
    { ... }
}
```

Listing 1. File states in Plaid

3. Typestate-Oriented Programming

In this section, we describe the typestate-oriented programming paradigm through a series of illustrative examples of files, collections, and iterators. Section 4 then demonstrates the generality of the approach through a richer example taken from the domain of GUI frameworks.

Our examples are written in Plaid, a typestate-oriented programming language currently under development at Carnegie Mellon University[3]. To aid in readability, Plaid uses Java's syntax wherever possible.

3.1 States

Consider the example in Listing 1 which declares the interface of a `File` in terms of `OpenFile` and `ClosedFile` states. These abstractions are declared using the **`state`** keyword; states are just like Java's classes, except that the state of an object may change as an object evolves. We observe that while the `filename` property is present in all files, the `read` and `close` methods are only available in the `OpenFile` state, and the `open` method is only available in the `ClosedFile` state. Methods and fields are declared much as they are in Java; in fact, the declarations of `filename`, `filePtr`, and `read` are legal Java syntax. Here `filePtr` refers to some low-level operating system resource such as a

[3] `http://www.plaid-lang.org/`

```
int readFromFile(ClosedFile f) {
    openHelper(f);
    int x = computeBase() + f.read();
    f.close();
    return x;
}
```

Listing 2. File client in Plaid

FILE* in the standard C libraries, and is only present in the OpenFile state.

The open and close methods introduce the first real new bit of syntax, as we must specify that the receiver object transitions between the OpenFile and ClosedFile states. In Plaid, any argument that changes state is declared with a pair of before and after states separated with the >> symbol. For example, a function openHelper that opens a file might be declared as follows:

```
void openHelper(ClosedFile>>OpenFile aFile);
```

For the close and open methods, we are actually specifying a transition for the receiver **this**, which is passed implicitly in Java-like languages. We allow the developer to specify state changes on the receiver by placing the state change specification in brackets [] after the normal argument list.

3.2 Tracking State Changes

Consider the File client shown in Listing 2, which accepts a closed file, opens it using the openHelper function, reads an integer and adds a base number, closes the file, and returns the computed sum. The readFromFile function accepts a single argument, ClosedFile f, and since no state transition is specified we take this as syntactic sugar for ClosedFile>>ClosedFile f. In this case the function signature implies that, though the function may change the state of the file internally, by the end of the function the file is once again in the ClosedFile state.

The compiler tracks the state of f through the body of the function and alerts the programmer if it is used inconsistently. In this case the compiler notes that openHelper transitions f from the ClosedFile state to the OpenFile state. As a result the call to f.read() is legal; if we comment out the openHelper line then the call to f.read() results in a compiler error. Likewise, the compiler notes that the call to f.close() transitions f to the ClosedFile state, which is the correct final state according to the function signature. If we comment out f.close() then we get an error stating that f should be in the ClosedFile state, but it is actually in the OpenFile state.

3.3 Aliasing and Permissions

The call to computeBase in Listing 2 points out an important subtlety. What if we have stored an alias to f somewhere in a global variable or in the heap? In that case we have to consider the possibility that the call to computeBase might close the file and make the call to f.read() illegal.

We rule out this possibility using a system of *permissions* [Bierhoff and Aldrich 2007], which abstractly describe whether and how an object is shared. One of our permissions is **unique**, indicating that there are no aliases to the object referred to by a function argument, return value, or field. We can therefore declare f with the keyword **unique** before the specified state, as shown below:

```
int readFromFile(unique ClosedFile f);
```

Since the compiler knows there are no aliases to f, the compiler also knows that computeBase will not close the file, and thus the call to f.read() is OK. Because **unique** is convenient, we make it the default permission, so it is unnecessary to declare it specifically—so in fact, the declaration of f in Listing 2 is semantically equivalent to the more detailed declaration above.

All references need permissions, so in Listing 1, the filePtr field, as well as the **this** argument to all the methods shown, are implicitly **unique**. The filename field is interesting because it has type String, which is an *immutable* (unchangeable) type in both Java and Plaid. Aliasing generally only causes problems for mutable objects, so it is not a problem for fields of type String. Thus we should use a second permission kind, **immutable**, which unlike **unique** allows unlimited aliasing of the object, but prohibits the object from being changed in any way. We could thus declare the filename field as:

```
public final immutable String filename;
```

However, all Strings in the application are **immutable**, so we instead declare the whole String state **immutable** as follows:

```
immutable state String { ... }
```

The meaning of this declaration is that whenever we use the state String to declare a variable, that variable defaults to the **immutable** permission kind. Thus the code in Listing 1 does not need to be changed; the filename field defaults to **immutable** because String is an **immutable** state, while the cFilePtr field defaults to unique (presumably because CFilePtr is not an **immutable** state).

3.4 Instantiating Objects

Plaid provides a slightly different object construction model than does Java. A full discussion is out of scope here, but we provide a **new** expression form that specifies the state of the object to be created along with values for all its fields. The **new** expression requires all fields to be visible, and since fields tend to be **private** or **protected** we will typically provide a factory-style **static** method within a state that

clients can use[4]. For example, we might have the following inside ClosedFile:

```
public static ClosedFile create(String f) {
    return new ClosedFile { filename = f; }
}
```

Observe that according to our defaulting convention, the returned ClosedFile will be **unique**.

3.5 Implementing State Changes

How does the file get opened? We can define the open function in ClosedFile as follows:

```
public void open() [ClosedFile>>OpenFile] {
     this <- OpenFile {
                  filePtr = fopen(filename);
             }
}
```

In the code above, the expression form e <- S { decls } transitions the object described by e into the state S, and uses the declarations in decls to initialize the fields of S that were not already present in e's current state (and to assign new values to pre-existing fields). In this case, we transition the receiver into the OpenFile state. The OpenFile state has two fields, filename (inherited from File) and filePtr. The filename does not need to be re-initialized, because the receiver is already in a substate of File and so filename is already defined. However, we must provide a value for filePtr, which we do by calling into an imported C library function (declaration not shown).

3.6 Shared Objects

So far we have introduced **unique** and **immutable** permissions. Each facilitates reasoning—**unique** because there are no aliases, and **immutable** because there are no state changes—but many real programs contain mutable, aliased objects. As previously proposed in our typestate system [Bierhoff et al. 2009] we provide additional permissions such as **shared**, a permission that indicates sharing and mutation via the shared references.

It is more difficult to track the state of a **shared** File, because any function call could potentially access an alias to that file and change its state (e.g. by closing it) without the caller knowing. For this reason, we treat the state associated with each **shared** permission as a *state guarantee*, which is a state that all clients of the object agree to respect. For example, in the code below, the **shared** permission is guaranteed to remain in the OpenFile state:

```
void logToFile(shared OpenFile logFile){
    logFile.write(LOG_TEXT);
}
```

[4] In Plaid, a **static** method is really an instance method of the object representing the surrounding state, much like class methods in Smalltalk

```
state Collection {
    type TElem;
    public void add(TElem>>none e);
    public TElem removeAny();
    public void remove(none>>TElem e);
}
```

Listing 3. Collections in Plaid

Another approach for dealing with **shared** objects is dynamically testing the object's state before performing a sensitive operation. An example of a state test function is given in the Iterator example below. For a full discussion of reasoning in the presence of **shared** and related permissions, see [Bierhoff and Aldrich 2007].

3.7 Genericity

We would like to be able to store our files in a collection, and keep track of the fact that the files are **unique** (if they are) and whether they are open or closed. As we discussed in Section 2, an advantage of building states into the language is that we use a single parameterization mechanism rather than having one for types and a separate one for permissions.

Listing 3 shows the interface of a Collection state in Plaid. We declare an *abstract type* TElem in the Collection state [Milner et al. 1997]. Abstract types are like type parameters but more modular[5], and we provide a syntactic sugar where Collection<String> means that the TElem type member of the collection is bound to String. Since Plaid types include both a state and a permission, the abstract type represents both the permission (**unique**, **immutable**, or **shared**) and the state the elements are in (e.g. OpenFile vs. ClosedFile).

Because the TElem type might be **unique**, and **unique** references cannot be duplicated/aliased, we must be careful about managing permissions as objects are moved in and out of the collection. When we add an object, a TElem permission is required to the parameter e, and that permission is stored in the collection and is not returned to the client. This is symbolized by the transition PElem>>**none**, where the **none** keyword indicates the lack of a permission to the object.

The removeAny function removes an arbitrary element and returns it with a TElem permission. We can also remove a particular element, passing in the element as a parameter without any permission, and receiving a TElem permission back from the collection.

[5] it allows a client to refer to Collection without specifying what TElem is if the element type doesn't matter to that client

An example client might look like this:

```
public void collectionClient(
            Collection<unique OpenFile> c) {
    unique ClosedFile aFile
             = ClosedFile.create(aFilename);
    aFile.open(); // aFile is now an OpenFile
    c.add(aFile);
    ... // cannot call aFile.read—no permission
    c.remove(aFile); // permission restored
    int readValue = aFile.read();
    ... // use readValue
}
```

In our example all functions either add or remove an element from the collection. There is no way to get an element while leaving it in the collection, because that would create an alias and if `TElem` was a **`unique`** permission then the invariant that **`unique`** permissions are not aliased would be violated. In contrast, we say that **`immutable`** and **`shared`** are *conserved* because they can be duplicated, resulting in 2 identical permissions. It is safe to define a `getAny` function that operates only on **`conserved`** permissions:

```
<TThis extends
     unique Collection<conserved TElem>>
public TElem getAny()[TThis];
```

In the example above, `TThis` is a type parameter of the method `getAny`. The type parameter has a bound stating that it must be a **`unique`** permission to some kind of `Collection` where the `TElem` is **`conserved`**. The type parameter is the bound for the receiver object **`this`**.

3.8 Example: Iterators

Listing 4 shows how an `Iterator` abstraction can be defined over collections. We add an `iterator` method that returns an iterator object. This method requires a permission `TThis` to the receiver object **`this`**. We want that permission to be **`immutable`** to enforce the common constraint (e.g. from Java and C#) that collections cannot be modified while they are iterated over.[6] As with `getAny`, we need the `TElem` permission to be **`conserved`**.

The state `Iterator` holds an **`immutable`** reference to the collection that it is iterating over and the states `Avail` and `End` are representative of the states that an iterator can be in, as shown in Figure 1. The `next()` method of the `Avail` state changes the state of the iterator from `Avail` to `End` if there are no more elements in the collection, otherwise the state remains `Available`.

Listing 5 shows how a client may use an iterator in Plaid. The usage of the iterator relies on the current state of the variable `i`. We only call `i.next()` after checking if `i` is in the `Avail` state by using the keyword **`instate`** (similar to **`instanceof`** in Java).

[6] of course, we want to regain a **`unique`** permission to the collection after iteration so we can modify it at that point. Our solution to this problem is outlined in [Bierhoff 2006].

```
state Collection {
    ...
    <TThis extends
      immutable Collection<conserved TElem>>
    public Iterator<TElem> iterator()
                            [TThis>>none];

}

state Iterator {
    conserved type TElem;
    final immutable Collection<TElem> coll;
}

state Avail extends Iterator {
    TElem next() [Avail >> (Avail || End)];
}

state End extends Iterator {
}
```

Listing 4. Iterators in Plaid

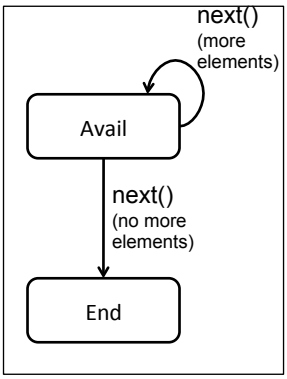

Figure 1. Iterator state machine

```
Collection<String> c = ...
Iterator<String> i = c.iterator();

while(i instate Avail) {
  String o = i.next();
}
```

Listing 5. Iterator client code in Plaid

4. Typestate-Oriented Design: Interactors

Files, collections, and interators make good explanatory examples due to their familiarity, but one may ask if the idea of states applies in other domains, a question we investigate here.

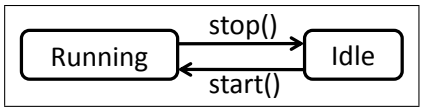

Figure 2. Interactor state diagram.

```
state Idle {
  void start() [Idle >> Running];
}
state Running {
  void stop() [Running >> Idle];
  void run(InputEvent e);
}
state MoveIdle extends Idle {
  GraphicalObject go;
  void start() [Idle >> Running] {
    this <- Running {
      void run(InputEvent e) {
        go.move(e.x,e.y);
      }
      void stop() [Running >> Idle] {
        this <- MoveIdle{}
      }
    }
  }
}
```

Listing 6. Interactor code in Plaid.

Graphical user interfaces (GUIs), like simulations, were one of the earliest applications of object-oriented programming [Kay 1993]. We believe typestate-oriented programming can extend the power of objects in this domain. To illustrate this, we discuss *interactors*, an influential concept in the GUI framework community. Interactors are reusable plugins for graphical user interface frameworks that handle user input [Myers 1990]. Each interactor encapsulates a high-level interactive behavior (e.g selection, rotation) and abstracts away underlying events (e.g. mouse movement, key click). Interactor methods are called by the framework when user input events are received. A developer adds an interactor to her graphical user interface to easily add behavior to graphical elements. For example, let's say she is developing a drawing tool and wants all drawn objects to be movable. With interactors, she only needs to add a (probably predefined) move interactor object to the window.

The interactors we discuss in this example are modeled with the state diagram shown in Figure 2. All interactors are either idle or running. An idle interactor transitions to the running state when the `start` method is called, and the `stop` method causes the opposite transition.

The code in Listing 6 defines two states, `Idle` and `Running` with undefined methods akin to Java interfaces. The `start` method transitions the receiver from the `Idle` state to the `Running` state and `stop` acts correspondingly. The `run` method does not transition the receiver. Instead, the interactor acts on the graphical object for which it is responsible.

An idle move interactor is represented as the `MoveIdle` state. This interactor creates a running move interactor when started. The running interactor is in an anonymous substate of `Running`. When the `run` method is called, the object invokes the `move` method on the associated graphical object. The `stop` method transitions the object back to the `MoveIdle` state.

As mentioned earlier, the GUI framework calls the appropriate interactor methods in response to events. For example, the `run` method of a move interactor is called when the mouse moves. Notice that a call to the `run` method does not change the interactor's state. Assume the framework stores the list of currently running interactors and passes incoming events to them. In Plaid, if the framework calls `run` on an interactor, the framework is *guaranteed* to be able to call `run` again when the next event fires. In Java, on the other hand, a developer who writes a new interactor might erroneously create a `run` method that stops the interactor under certain conditions. In that case the Java framework would have to either perform a dynamic state check before every call to the `run` method, or throw an exception, neither of which is desirable.

5. Challenges

In order to further investigate the typestate-oriented programming paradigm, we are currently working on an interpreter and typechecker for the Plaid language described in this paper.[7] Many challenges remain, however, in making the vision described here a reality:

Overhead Because we are documenting and tracking more sophisticated properties in our system than is typical for type systems, a major open question is whether the approach will be practical, or whether the conceptual overhead or the overhead of declaring our permissions will be too high. It is promising that the examples in this paper are not substantially more complex than the equivalent Java code.

Sharing We briefly described the challenges of tracking object state for **`shared`** permissions. Prior work has shown that ideas like state guarantees and dynamic state test functions can provide great leverage, but we expect we will need new ideas in this area as well. Experience will show whether it is feasible to do the kind of reasoning envisioned in this proposal on ordinary code.

Extensions The language described here is very simple. While we hope Plaid remains straightforward, new features will be needed to make the language usable in industrial practice. Some of these will present challenges; for example, will adding concurrency make it more difficult to track typestate? At the same time, we hope that

[7] Available at `http://www.plaid-lang.org/`

the ideas in Plaid will provide synergies as well. For example, our companion Onward! paper examines how the kind of permissions used here could make concurrency easier and safer [Stork et al. 2009].

6. Conclusion

This paper presented a new paradigm, typestate-oriented programming, which introduces states as first-class abstractions for describing object interfaces and representations. Interfaces can more directly express, and the type system can enforce, constraints that are only implicit in most object-oriented languages, such as when it is legal to read from a file or call `next` on an iterator. The representation of objects can change in a natural way when their states do, for example allowing an `OpenFile` to hold a low-level file resource that is not present in closed files. By integrating states into the language, we can provide this additional expressive power while minimizing added complexity, for example by using a single parameterization mechanism rather than one each for states, permissions, and types. Our Interactors example demonstrates that the idea of states is not specific to common library abstractions like files and collections, but applies in GUI frameworks (and we expect other domains) as well.

Much work remains to be done before making the vision of typestate-oriented programming a practical reality, and we are actively investigating both the theoretical underpinnings of the paradigm and the practical tradeoffs involved in building a real language, Plaid. However, we believe that states are a fundamental programming abstraction, and that the future of direct language support for states is bright.

Acknowledgments

This work was supported in part by DARPA grant #HR0011-0710019, NSF grants CCF-0546550 and CCF-0811592, and Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems." We thank Eric Tanter and the Plaid group for their helpful feedback on earlier versions of this paper.

References

Andi Bejleri, Jonathan Aldrich, and Kevin Bierhoff. Ego: Controlling the Power of Simplicity. In *Proc. Foundations of Object-Oriented Languages*, 2006.

Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. A Mechanism for Flexible Dynamic Trait Replacement. In *Proc. Formal Techniques for Java-like Programs*, 2009.

Kevin Bierhoff. Iterator Specification with Typestates. In *Proc. Specification and Verification of Component-Based Systems*, 2006.

Kevin Bierhoff and Jonathan Aldrich. Modular Typestate Checking of Aliased Objects. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API Protocol Checking with Access Permissions. In *Proc. European Conference on Object-Oriented Programming*, 2009.

Eric Bodden, Laurie Hendren, Patrick Lam, Ondrej Lhotak, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. *Oxford Journal of Logics and Computation*, 2008.

Lera Boroditsky. How Does Language Shape the Way We Think? In Max Brockman, editor, *What's Next? Dispatches on the Future of Science*, pages 116–129. Vintage, 2009.

Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based Simulation Language. *Communications of the ACM*, 9(9):671–678, 1966.

Robert Deline and Manuel Fahndrich. Typestates for Objects. In *Proc. European Conference on Object-Oriented Programming*, 2004.

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Re-classification. In *Proc. European Conference on Object-Oriented Programming*, 2001.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. *Transactions on Software Engineering and Methodology*, 17(2), 2008.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence*, 1973.

Alan C. Kay. The Early History of Smalltalk. *SIGPLAN Notices*, 28(3), 1993.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

Brad A. Myers. A New Model for Handling Input. *ACM Transactions on Information Systems*, 8(3):289–320, 1990.

Barbara Pernici. Objects with Roles. In *Proc. Conference on Office Information Systems*, 1990.

Asher Sterkin. State[chart]-Oriented Programming. In *Proc. Multiparadigm Programming with Object-Oriented Languages*, 2008.

Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *Proc. Onward!*, 2009.

Robert E Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

Antero Taivalsaari. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming*, 6(3):25–32, 1993.

David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 1987.