1 Introduction

If we intend to have more examples that target the existing language, it would be useful to clarify the goals and target audience, and feed this information into the roadmap for developing the language and implementation. For example, compatibility might be irrelevant or important. (It might be important if we want to make the idea more plausible, or if we want to get some feedback from actual users.) There are implications: for example aiming for compatibility would suggest using annotations instead of new bespoke syntax if possible.

As another example, are we specifically interested in communication? If so, are we only interested in communication over sockets? Is the goal to demonstate that typestate can be used for typing channels, or that the resulting programming model is convenient and user-friendly? Again, there are implications for how we design the language.

2 Typestate definitions

Early typestate language such as Plural [1] used pre- and post-condition annotations on methods to specify state transitions. More recent languages such as Plaid [3] unify states with classes; in these "typestate-oriented" languages, state change is object reclassification. Our approach is different again: we introducing a separate typestate definition. This may have merits, but it also has a significant cost in requiring method and type signatures to be duplicated. Both the user, and the implementation, are forced to consider the interaction between typestate declarations and other Java definitions. For example:

- Overloading. What if I define two methods called `buy`?
- Mismatches between typestate definitions and Java definitions.
- Extending typestate with new features, e.g. inheritance or exceptions.
- Integration with IDE features like refactoring.

An alternative approach based on annotations is outlined below.

2.1 Representing typestates using interfaces and annotations

An interface specifies the methods available on an object. Since this is what a typestate also does, a natural question is: can a typestate be captured as an interface where each method is annotated with a specification of the resulting state? In this approach, tracking the state of an object statically is like implicitly casting it between the various state interfaces; the underlying implementation class has to implement all the methods and all the interfaces. (There is no multiple inheritance in Java, which is why we propose interfaces rather than classes.)

First we use a so-called *annotation type* to define a new kind of Java annotation called `Target`:

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
public @interface Become {
   Class<? extends Object> value ();
}
```

`Target` allows us to attached annotations of the form `@Become(State.class)` to a method, specifying `State` as the name of the interface representing the typestate of the object after the method invocation. We can use these annotations to decorate the interface methods, in a backwards-compatible way, with Plaid-style transition specifications [3]. The following example illustrates; the `@ResultCase` construct is explained below.

```
public interface Buyer {
   interface Init {
      @Become(Shop.class)
      void init (Channel u);
   }

   interface Shop {
       @Become(Shop.class)
       Price price (Product p);

       @ResultCase(is = Result.OK, then = @Become(Pay.class))
       @ResultCase(is = Result.ERROR, then = @Become(Shop.class))
       void buy (Product p);

       @Become(End.class)
       void stop ();
   }

   interface Pay {
       @Become(Shop.class)
       void pay (Payment p);
   }
}
```

(The enclosing `Buyer` interface is acting as a namespace, rather than a state interface.) The implementation is essentially as it was before, except that the class `BuyerImpl` must implement each of the state interfaces. The job of the typestate extensions to the compiler is to allow the user to call methods on the state interfaces without having to explicitly cast. Java Specification Request (JSR) 269 [4] adds a pluggable annotation processing API to Java to allow the authoring of tools that process annotations; these facilities can be accessed via command-line options to `javac`. The open source Checker framework [5, 2] is a comprehensive example of the use of annotations to extend Java's type system.

```
class BuyerImpl implements Buyer.Init, Buyer.Shop, Buyer.Pay {
```

```
    public void init (Channel u) { ... }
    public void pay(Payment p) { ... }
    public Price price (Product p) { ... }
    public void buy (Product p) { ... }
    public void stop() { ... }
}
```

The duplicate annotations like this are legal in Java 8. Because annotation types can't make use of generics, the user would also have to define

```
@interface ResultCase {
    Result is ();
    Become then ();
}
```

References

[1] K. Bierhoff, N. Beckman, and J. Aldrich. Checking Concurrent Typestate with Access Permissions in Plural: A Retrospective. In P. L. Tarr and A. L. Wolf, editors, *Engineering of Software*, pages 35–48. Springer Berlin Heidelberg, 2011.

[2] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 681–690, New York, NY, USA, 2011. ACM.

[3] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, Oct. 2014.

[4] Java Community Process. JSR 269: Pluggable Annotation Processing API. `https://www.jcp.org/en/jsr/detail?id=269`, 2014.

[5] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.