

Typestate Protocol Specification in JML

Taekgoo Kim

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
vanang@cs.cmu.edu

Kevin Bierhoff

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
kevin.bierhoff@cs.cmu.edu

Jonathan Aldrich

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

Sungwon Kang

Dept of Computer Science
KAIST
119 Munjiro Yuseong-gu
Daejeon, 305732, Korea
sungwon.kang@kaist.ac.kr

ABSTRACT

The Java Modeling Language (JML) is a language for specifying the behavior of Java source code. However, it can describe the protocols of Java classes and interfaces only implicitly. Typestate protocol specification is a more direct, lightweight and abstract way of documenting usage protocols for object-oriented programs. In this paper, we propose a technique for incorporating the typestate concept into JML for specifying protocols of Java classes and interfaces, based on our previous research on typestate protocol specifications [4]. This paper presents a set of formal translation rules for encoding typestate protocol specifications into pre/post-condition specifications. It shows how typestate protocol specifications can be mixed with pre/post-condition specifications and how violations of code contracts in inheritance can be handled. Finally, our proposed technique is demonstrated within the Java/JML environment to show its effectiveness.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specification-Languages; D.2.4 [Software Engineering]: Software/Program Verification; D.2.2 [Software Engineering]: Design Tools and Techniques; F.3.1 [Theory of Computation]: Specifying and Verifying and Reasoning about Programs

General Terms

Specification, Verification, Design, Language

Keywords

Typestate, JML, behavioral subtyping, usage protocol.

1. INTRODUCTION

As the size of a software system grows, the likelihood of errors in that system becomes much greater. Much of this growth comes from errors due to inconsistencies between the intended and actual use of components within the system. For example, a

programmer must follow the contract of a method, meaning that a client of a particular class or interface should follow proper method call sequences as well as the usage rules of each method. When the programmer calls methods in the wrong order or violates other usage rules, the method cannot guarantee anything about the result, and in fact may produce erroneous side effects like runtime exceptions or program failure. For example, trying to read data from a closed Reader stream in the Java IO Library may result in an IO exception being thrown, causing the application to fail. In practice, numerous APIs have implicitly protocols [16] such as JDBC and other Java libraries. Thus, there is a need for an explicit way to document and enforce the contract of a method.

One way of addressing this issue is to formally specify component interfaces within the software system and ensure that clients follow the specification [5]. For example, Hoare proposed a formal specification methodology based on using pre- and post-conditions to specify the usage protocol of a component [8]. The Java Modeling Language (JML) supports this ‘design by contract’ methodology in the context of Java [1]. For instance, the contracts can be defined within program code as annotations for member functions or variables, and can be translated into executable code by a JML compiler. While the JML program is running, any violation of the contract can be detected by a JML run-time checker.

Pre- and post-conditions in JML can be used to precisely describe the usage protocols of Java classes and interfaces. However, in this case the usage protocol is not defined in terms of explicit states and transitions, but rather in terms of predicates on the object’s state before and after the method. Inferring how different methods relate, and the legal sequences of calls to those methods, can therefore be done only indirectly. Thus, although the pre-/post-condition specification technique is very powerful, it is not always the most direct or easy to understand way to express a usage protocol.

Typestate is a lightweight and abstract way of presenting usage protocols [6]. The concept behind typestate is to define a state machine made up of a number of explicit states, where each method in the class transitions the receiver object from one state to another. Therefore, typestate is a natural and direct way to express usage protocols, but because the states are by their nature abstract and finite, it cannot be used to specify behavior in as much detail as a pre-/post-condition style specification can.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAVCBS’09, August 25, 2009, Amsterdam, The Netherlands.

Copyright 2009 ACM 978-1-60558-680-9/09/08...\$10.00.

In this paper, we propose to use `typestate` to specify and verify the behavior of a type (i.e. an interface or a class), based on previous research on `typestate` protocol specifications [4]. This paper makes the following contributions:

- We propose an extension to the syntax of JML that supports expressing `typestate` protocols directly.
- We propose a set of translation rules from this `typestate` protocol definition syntax to standard JML syntax, both providing a formal definition for the semantics of our extension, and providing a guide to the implementation of the system.
- Our design supports mixing `typestate` protocols and pure JML specifications, so developers can specify behavior in a lightweight way with `typestate` protocols, and seamlessly extend that specification with more heavyweight traditional JML specifications.
- Our design can support safe reasoning about flexible uses of inheritance, where a subclass may have internal representation invariants that are incompatible with the representation invariants of superclasses (Section 5.1.3).
- We validate our design by using our `typestate` protocol specifications on example code, translating those specifications (by hand, for now) to JML, and using existing JML tools to verify the code against those specifications.

The remainder of this paper is organized as follows: Section 2 presents previous research related to `typestate` protocol specification; Section 3 extends the pre-existing JML syntax with new constructs to express `typestate` protocols. In Section 4, we present a set of corresponding translation rules from the new `typestate` syntax to existing JML syntax. A case study based on a simple Java application is presented in Section 5; we summarize our work and conclude in Section 6.

2. RELATED WORK

Hoare suggested a formal methodology that provides a set of rules to reason about the correctness of a program using mathematical logic. His method is based on the idea of a specification as a contract between the implementation and its clients, where the specification consists of pre/post-conditions and invariants of the software system [8]. By writing explicit pre/post-conditions and invariants, one can verify that a client follows the usage protocol of a component, and then reduce mistakes that cause system failure.

State-based specification methods such as Z [3] can be used for specifying systems as well. Object-Z [12] adapts Z to object-oriented systems. It can capture class invariants and supports pre- and post-conditions of methods. However, Object-Z has no immediate mapping onto an implementation.

Typestates were initially proposed for imperative languages [6]. DeLine and Fähndrich proposed typestates for objects [7], as embodied in the Fugue language. Fugue allows subclasses to define additional states. Classes can define predicates that describe states in terms of instance fields. Bierhoff and Aldrich [4] modify Fugue’s approach with the concepts of state refinement, which ensures subtype substitutability, and specification inheritance similar to the JML, which ensures

behavioral subtyping. Our design builds on that of Bierhoff & Aldrich. Butkevich et al. describe protocols as labeled transition systems, check dynamically for protocol usage violations, and can statically check for hierarchy violations [10]. Barnett, Rustan, Leino and Schulte introduce Spec# [14], a formal language for API contracts similar to JML and Eiffel [11]. Spec# extends C# with constructs for code specification and reasoning about object invariants. Also, it has unique features for maintaining invariants in the presence of callbacks, threads and inter-object relationships. Cheon and Perumendla extend JML to specify protocol property of program modules that allow developers to specify the sequences of method calls in a process algebra-style [15]. However, this method have serious scalability problem because there is no way to handle state dimensions.

3. TYPESTATE PROTOCOL SPECIFICATION IN JML

In this section, we introduce extensions to the syntax of JML for specifying `typestate` protocols. Our protocol specifications are comprised of 4 parts: state definitions, state invariants, protocol specifications, and state tests. In first two subsections of this section, we describe how abstract states can be defined and given semantics in terms of implementation predicates. The next two subsections show how protocols can be defined with these states and how JML specifications can test the state of an object. In the final subsection, we present a solution for describing frame axioms. We discuss the strategy for encoding our `typestate` protocol syntax into existing JML constructs in Section 4.

3.1 Defining States

Figure 1 presents the syntax for defining a finite set of conceptual states within a type. We follow Bierhoff and Aldrich in defining new states as refinements of an existing one, a choice which facilitates behavioral subtyping, since a type in the new state is a behavioral subtype of the same type in the state that was refined. The refined state could have been declared either in the current class or a superclass. A single state can be refined multiple times, which corresponds to orthogonal state dimensions [4] or AND-states in Statecharts [13]. State dimensions let us focus independently on different aspects of an object, for example on whether a file is open or closed independent of whether it is writeable or read-only. State definitions are marked with the keyword `state`.

The grammar defines a list of states as refinements of some existing state, which defaults to a global `alive` state. The optional `as` clause defines the name of the state dimension, which defaults to a predefined `default` dimension. The grammar allows a developer to put the state dimensions into a user-defined JML data group.

```

variable-decls ::= ... | state-decl
state-decl ::= state state-list [refine ident] [as ident] ; [jml-data-group-clause]
state-list ::= ident | state-list, ident

```

Figure 1: Grammar for State Definitions

3.2 State Invariants

Following Fugue, we define the semantics of an abstract state in terms of a predicate over the instance fields of the class. Semantically, whenever an object is in a particular state `s`, the

state invariant for `S` must be true. The syntax for defining state invariants is given in Figure 2.

```
jml-declaration ::= ... | modifiers state-invariant
state-invariant ::= state ident <==> predicate ;
```

Figure 2: Grammar for State Invariants

3.3 Protocol Specifications

In tystate protocol specifications, protocols are defined with state transitions that define the pre- and post-conditions of a method in terms of states. For a given method, a developer can define multiple transitions, called specification cases of the method. In our syntax, as in JML's, specification cases are separated with the keyword `also`. `also` can also be used for indicating that the specification of a supertype's method with same name should be inherited. In tystate protocols, state transitions are introduced with the keyword `protocol`.

```
simple-spec-body-clause ::= ... | protocol-clause
protocol-clause ::= protocol protocol-product -> protocol-union
protocol-union ::= protocol-product | protocol-union ' | ' protocol-union
protocol-product ::= predicate | (predicate, ..., predicate)
```

Figure 3: Grammar for Protocol Specifications

Figure 3 shows the grammar for protocol specifications. A protocol-clause concisely defines a pre- and post-condition pair. A product notation (p, q, \dots) (where p and q are predicates) defines multiple conjunctive conditions, increasing readability. Predicates within the product are boolean expressions, and will usually include state tests (see below).

3.4 State Tests

A state test is a predicate testing whether an object is currently in a particular state. State tests will be used for defining protocols, but can be used anywhere a predicate can appear in JML. Figure 4 shows the syntax for state tests.

```
relational-expr ::= ... | shift-expr \in ident
```

Figure 4: Grammar for State Tests

Within JML, the state test can be treated as a `relational-expr`, and has the same precedence as the other relational operators.

We only allow testing the state of an object against a constant with `\in`. Due to the difficulty of encoding state tests in the presence of subtyping, a comparison of states between objects is left to future work.

3.5 Assignables

The encoding of protocols is treated orthogonally to JML's assignable clause. It is up to the developer to specify what data groups can be assigned in a given method. However, the developer has to be aware that states are mapped hierarchically into a separate data group `alive`. Thus, if assignable clauses are defined, one must ensure that states can be changed as desired. The easiest way to accomplish this is to include `alive` into the list of assignable data groups. One can be more precise, though, by limiting possible changes to a substate or dimension. Notice, however, that such a restriction limits flexibility of overriding

methods to change states within new or unrelated state dimensions.

Beyond this simple solution, data group mappings between states and other fields can be defined. We already discussed in section 3.1 that state dimensions can be mapped into arbitrary data groups besides `alive`. Conversely, concrete and model fields can be mapped into a state's or dimension's data group.

4. TRANSLATING TO PURE JML

In this section, we present formal rules translating tystate protocol specifications into standard JML, covering state definitions, the invariants associated with those states, protocol specifications, and state tests.

4.1 Translation Rules for State Definitions

In JML, a specification field can be declared with `model` or `ghost` modifiers [2]. Likewise, one can declare states as `model` or `ghost` states with appropriate modifiers. In addition, the developer can also limit the visibility of states with modifiers such as `private`, `protected` and `public`. This is supported because the declaration of tystates syntactically extends JML variable declarations (`variable-decls`, see Section 3.1).

However, only some Java and JML modifiers are meaningful in the context of state definitions. For example, modifiers such as `public model` or `private ghost` are meaningful modifiers in the context of a state definition, whereas JML modifiers `native` and `pure` are not. We allow the following modifiers on states:

- `model/ghost` (JML modifier). These modifiers prescribe a translation into `model` or `ghost` fields. A `model` field is an abstraction of one or more concrete fields. Thus, `model` states must be accompanied by a `represents` clause that defines whether the object is in that state in terms of concrete Java fields. The `ghost` field is similar to `model` field in terms of its purpose for defining a specification-only field, but the value of a `ghost` field is determined by its initialization or `set-statement` in a method body rather than determined by a `represents` clause. Therefore, an object transitions from one `ghost` state to another by assigning boolean values to the corresponding `ghost` field in method bodies. Because the implementation predicate in a state definition can refer to any concrete field as well as `ghost` fields, we treat `model` as a default modifier in state definitions.
- The Java visibility modifiers `private`, `protected` and `public` work in the same way as declaration of Java variable. Therefore, `private` states are not visible in clients or subtypes, `protected` states are visible in subtypes, and `public` allows visibility in both clients and subtypes.
- The `static` modifier on a state describes properties of the type and its static fields, not the instance state of that type. The `instance` modifier (the default) is the converse.
- The `final` modifier on a state prohibits refining that state further.

Figure 5 shows the rule for translating state definitions. Each declared state turns into a boolean field with the same name, with the semantics that the field's value is true exactly when the object is in the given state. If a dimension was specified, it is declared

as a JML data group, and the state fields are placed into that data group. The new JML data group is nested within the superstate that is being refined, as well as the data group G (if specified in the typestate declaration). If no dimension was specified in the declaration, we create a fresh data group to represent the dimension internally. If no superstate was specified, we use the alive state (root state).

```
[modifiers state S1, S2, ..., Sn (refine S) (as D); (in G;)]
    ==>
modifiers non_null model JMLDataGroup D; in S(, G);
modifiers boolean Si; in D; ... modifiers boolean Sn; in D;

modifiers invariant S ==> ( S1 || S2 || ... || Sn );
modifiers invariant S1 ==> S...modifiers invariant Sn ==> S;

S: Supertype's state
Si: Subtype's states refined from the supertype's state
D: State dimension
```

Figure 5: Rule SD for State Definitions

The refinement relationship between a state and its refined states is defined using invariants, as shown. In particular, if the object is in the superstate, then it must be in one of the substates. Furthermore, if an object is in any substate, then it must be in the superstate as well. Although semantically we view an object as being in exactly one state, our translation strategy does not enforce this (e.g. by using exclusive or) because the end user may sometimes want to overapproximate the conditions under which an object is in a particular state, e.g. to avoid using very complex predicates.

```
public model state open, closed refine alive as mode;
    ==>
public non_null model JMLDataGroup mode; in alive;
public model boolean open; in mode;
public model boolean closed; in mode;
public invariant alive ==> (open || closed);
public invariant open ==> alive;
public invariant closed ==> alive;
```

Figure 6: Example of a translation by Rule SD

Figure 6 shows an example translation. We declare two states open and closed, which are refined from the root state alive, and assign those states to dimension mode.

4.2 Translation Rules for State Invariants

Our translation strategy for state invariants is relatively straightforward. State invariants are only used in the case of model states. It is unnecessary to impose state invariants to ghost states because a ghost field by definition does not have a value determined by concrete fields. Rather, its value can only be set by the set statement ([2], p.11) in method bodies.

```
[modifiers state S <==> B;]
    ==>
modifiers represents S <- Ssuper && [B];

Ssuper: Supertype's state which has invariants for itself.
B: State invariant for S
```

Figure 7: Rule SI for State Invariant

On the other hand, a model field should be represented by concrete fields. As shown in Figure 7, we use a JML represents clause with a left arrow ('<-') to map the model field for the state to the state invariant expression. As with

modifiers for state definitions, modifiers for state invariants are preserved in translation. A developer can use any fields (concrete, model and ghost) in the boolean state invariant expression B.

Note that our translation conjoins the field for the superstate S_{super} with the state invariant, to ensure by construction that the state invariant is never true unless the invariant for the superstate is true as well.

Figure 8 shows an example in which the open state has been refined into forward and backward states. As described, the model field for the open state must be conjoined with the state invariants declared for each substate.

```
public state forward <==> isForward;
public state backward <==> !isForward;
    ==>
public represents forward <- open && isForward;
public represents backward <- open && !isForward;
```

Figure 8: Example of Translation by Rule SI

4.3 Translation Rules for Protocol Specifications

The protocols for methods are straightforwardly encoded into pairs of requires and ensures clauses. In protocol specifications, a pre-state which is on the left-hand side of the transition notation '->' is encoded into a JML requires clause, whereas a post-state on the right-hand side of '->' is directly translated into an ensures clause.

Note that the predicate in the protocol-product can be an arbitrary JML predicate expression, so that the typestate protocol specification allows using state tests as well as state names. Since the ',' in the protocol-product means boolean AND between two predicates, the product notation (predicate₁, predicate₂, ..., predicate_n) should be encoded into the conjunction of each predicate with the logical operator '&&'. In translating the disjunction of two protocol-unions, we convert the boolean OR ('|') into '||', because in JML specifications '|' is used for bitwise or and '||' is used as the disjunctive logical operator.

```
[protocol protocol-product -> protocol-union]
    ==>
requires [protocol-product];
ensures [protocol-union];
```

Figure 9: Rule PS for Protocol Specification

```
['(predicate1, predicate2, ... predicaten)']
    ==>
[predicate1] && [predicate2] && ... && [predicaten]
```

Figure 10: Rule PP for Protocol Product in Figure 9

```
[protocol-union '|' protocol-union]
    ==>
[protocol-union] || [protocol-union]
```

Figure 11: Rule PU for Protocol Union in Figure 9

Figure 12 illustrates how protocol specifications can be translated into conventional JML specifications. The pre-state of the read method is open, and its post-state is still open. The also in the header part of the specification is a JML keyword to preserve overridden method's specification. In this example, read and

close methods override the corresponding methods of a supertype while preserving the super method's specification. Semantically, the pre/post-conditions of overridden methods will be conjoined with the contracts declared on overriding methods in the JML specification to maintain behavioral subtyping [9].

To illustrate case-by-case specifications, we have (perhaps unrealistically) given two protocol cases for the close method: an open stream is closed, while a closed stream remains closed. If multiple protocol clauses are present (and not separated by also) then we wrap their individual translations with a pair of brackets, '{|' and '|}'. In JML ([2], p.73), these brackets are used for nested specification cases, and the keyword also is used to join multiple specification cases. Likewise, multiple state transition cases are encoded with a pair of brackets and the keyword also inside the brackets to separate each case.

```

/*@ also
  //@ protocol open -> open
  public int read(char[] cbuf, int off, int len) throws
  IOException

  //@ also
  //@ protocol open -> closed;
  //@ protocol closed -> closed;
  public void close() throws IOException
  ==>

  //@ also
  //@ requires open;
  //@ ensures open;
  public int read(char[] cbuf, int off, int len) throws
  IOException

  //@ also
  //@ {|
  //@ requires open;
  //@ ensures closed;
  //@ also
  //@ requires closed;
  //@ ensures closed;
  //@ {|}
  public void close() throws IOException
  
```

Figure 12: Example of Translation of Protocol Specification

4.4 Translation Rules for State Tests

The developer can use state tests to check whether a type is in a certain state *S*. The translation of a state test is done by substituting the tpestates keyword \in with '.' operator of the JML specification (see Figure 13). In JML, the dot ('.') operator is used to refer to a field of the structure that the name followed by the operator refers to. To avoid confusion with in in the JML specification, we just add '\ ' as a prefix of in. The shift-expr of the JML specification allows the developer to test any form of a type such as 'this \in open' or 'fun.getType() \in closed'.

```

[shift-expr \in S]
                               ==>
shift-expr.S
  
```

Figure 13: Rule ST for State Test

Figure 14 demonstrates an example translation of a state test. Here \result is a JML keyword that represents the return value of the specified method.

```

protocol (\result ==> this \in closed) && (!\result ==>
this \in open);

                               ==>

ensures (\result ==> this.closed) && (!\result ==>
this.open);
  
```

Figure 14: Example of Translation of State Test

5. CASE STUDIES

In this section, we present case studies that demonstrate tpestate protocol specifications and their translation into JML. In Section 5.1, we use classes from the Java I/O library. Basic tpestate protocols are presented and the application of our technique to handle specification subtyping is also examined. Section 5.2 then demonstrates how tpestate protocol specifications can be freely mixed with specifications in ordinary JML.

All of the examples in this section are excerpts from code that compiles with the JML toolset, and which can be used to find protocol violation errors using the JML run time checking tools.

5.1 Java Readers: subtyping and refinement

The Reader class is similar to the InputStream class in Java, but it works with characters rather than with bytes. The read() method was designed to read a single character at one invocation and return the character as an integer ranging between 0 and 65535, or -1 when the end of stream is reached. The close() method closes the reader class and releases any resources associated with it. Thus, if one invokes the close() method, then read() cannot be called. In the following subsections, we demonstrate tpestate protocols and subtyping and state refinement with subclasses of Reader class.

5.1.1 Translation of Tpestate Specification for ScreenReader Class

Consider a ScreenReader class which extends the Reader class from the Java I/O library. This class operates like the Reader class but it screens out every occurrence of a certain character. For this case study, we defined two states for the ScreenReader class: open and closed. When the client calls read(), the state of the object must be open. The object enters the closed state when the close() method is invoked. Figure 15 illustrates the protocol of the ScreenReader class using a UML2.0 state machine diagram.

In Figure 16, in order to define two states, open and closed, the root state must be defined beforehand. Because all objects in Java directly or indirectly inherit from the Object class and we define a top-level alive state for the Object class. The states of Object subclasses are then refined from this root state, alive. Then, we put open and closed states into the mode state dimension using the as clause. Because tpestates like open and closed are usually represented as model fields of the class, we define the values of these fields using concrete member variables of the ScreenReader class.

The code at the bottom of Figure 16 shows how this tpestate protocol specification is translated into ordinary JML. We defined the root state first and put mode into alive state. The mode state dimension contains a set of states, open and closed, declared in the next two lines. The represents clauses come from the declared

invariants for each state, and define under what conditions the object is in each state.

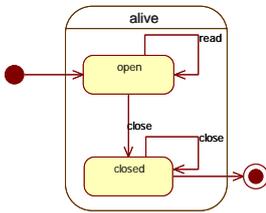


Figure 15: Protocol of ScreenReader class

```

public class ScreenReader extends Reader{
  //@ public model states open, closed refines alive as
  mode;
  //@ protected states open <==> income != null;
  //@ protected states closed <==> income == null;
  protected Reader income = null;

  //@ also
  //@ protocol open -> open
  public int read(char[] cbuf, int off, int len)

  //@ also
  //@ @ protocol open -> closed;
  //@ @ protocol closed -> closed;
  public void close()
  //@ protocol (\result <== this \in closed) &&
  (!\result <== this \in open);
  public boolean isClosed()
}

==>

public class ScreenReader extends Reader{
  //@ public model boolean alive;
  //@ public non_null model JMLDataGroup mode; in alive;
  //@ public model boolean open; in mode;
  //@ public model boolean closed; in mode;
  //@ public invariant alive ==> (open || closed);
  //@ public invariant open ==> alive;
  //@ public invariant closed ==> alive;
  //@ protected represents alive <- true;
  //@ protected represents open <- income != null;
  //@ protected represents closed <- income == null;
  protected Reader income = null;

  //@ also
  //@ requires open;
  //@ ensures open;
  public int read(char[] cbuf, int off, int len)

  //@ also
  //@ {}
  //@ requires open;
  //@ ensures closed;
  //@ also
  //@ requires closed;
  //@ ensures closed;
  //@ {}
  public void close()

  //@ ensures (\result <== this.closed) && (!\result <==
  this.open);
  public boolean isClosed()
}

```

Figure 16: Tpestate protocol specification of Figure 15

5.1.2 Translation of Tpestate Specification for a Subclass of ScreenReader

The ReversibleScreenReader class is designed to extend the ScreenReader class to reverse the order of characters read. When the buffer is opened, the client of this class can change the

direction of the stream to either forward or backward. The open state is refined into two states forward and backward. Thus, the protocols of this class can be defined with three states: forward, backward and closed. Figure 17 shows the protocol of ReversibleScreenReader. By invoking reverse() in Figure 18, the state of the object will be flipped between forward and backward. Like the ScreenReader class, the ReversibleScreenReader class will be closed when the close() is called. In this class, some methods such as read() and reverse() have multiple specification cases. For instance, read() method can be invoked in forward as well as backward, and afterwards it leaves the object in an unchanged state.

5.1.3 Handling Reuse Idioms

In Java programming, a developer sometimes creates new classes that inherit from existing classes, but violate the representation invariants of those superclasses. When a particular subtype overrides a method and violates the supertype's invariants, the refinement between the superstate and substates does not work. We designed the MemoryReader class to show that this can be a problem with tpestate protocols and how we handle this important issue.

The MemoryReader class extends the ScreenReader class but does not refine its attributes and behavior. Instead, this class has a string field that caches characters when the object is initially created. Also, the read method reads a character from its string field rather than reading one through the read method of the supertype. It seems the read method of the MemoryReader is overriding the one of the ScreenReader class, but in fact, the read method of the MemoryReader class does not inherit state representation invariants from its superclass.

Even though this class preserves the same state names of the superclass (open and closed), these states must have different state invariants. Here is an example of specification of the MemoryReader with tpestate protocols: when a developer instantiates the MemoryReader and uses the instance within a JML checker, the JML checker raises a post condition error for the constructor because the post-state must be the open state. In fact, because tpestate protocol on the JML conjoins state invariants of the ScreenReader for the open state, the open state of the MemoryReader preserves the state invariants of the ScreenReader. However, closing the buffer of the superclass at the end of the constructor violates the post-states.

Here we translate the tpestate protocol specification into a JML specification. Notice that we add two ghost boolean fields in the superclass. We add these fields anticipating the possible violations of code contracts by developers. Using the two given fields, developers can manipulate the state of the superclass as necessary.

The set clause in the constructor body in the ScreenReader is used to set a value for the declared ghost fields. Also, we explicitly disjoin a boolean ghost field with the existing state invariants. Therefore, the state invariant for open in the ScreenReader is now a disjunction of open_gh and 'income != null.' If open_gh is set to true, then the open state will be true regardless of the value of income.

However, the use of ghost field for handling violation of code contracts brings obvious disadvantages. First, the violation of code contract is handled in very ad-hoc fashion. In addition to,

this we anticipate extra burden of developers to track the `ghost` field to modify the state as necessary. The number of `ghost` field to take care can increase exponentially.

Beyond this ad-hoc solution, JML provides code modifier to handle this problem, which indicates a specification case containing methods are not inherited from its supertypes whereas the methods are overridden ([2], p. 121). Unfortunately, we remain this nicer solution with code modifier for next work until we get more obvious result and example.

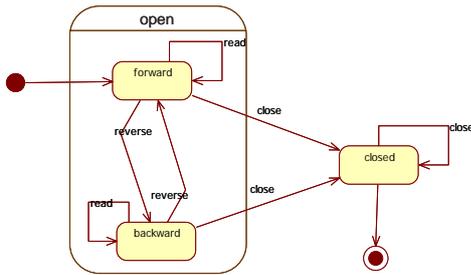


Figure 17: Protocol of ReversibleScreenReader

```
public class ReversibleScreenReader extends ScreenReader{
  //@ public model states forward, backward refines open
  as reverse; in mode;
  //@ public states forward <==> isForward;
  //@ public states backward <==> !isForward;
  private boolean isForward = true; // in \frame-
  local(reverse);

  //@ protocol forward -> backward;
  //@ protocol backward -> forward;
  public void reverse()
}

==>

public class ReversibleScreenReader extends ScreenReader{
  //@ public non_null model JMLDataGroup reverse; in
  open;
  //@ public model boolean forward; in reverse;
  //@ public model boolean backward; in reverse;
  //@ public invariant open ==> (forward || backward);
  //@ public invariant forward ==> open;
  //@ public invariant backward ==> open;
  //@ private represents forward <- open && isForward;
  //@ private represents backward <- open && !isForward;
  private boolean isForward = true;

  //@ {|
  //@ requires forward;
  //@ ensures backward;
  //@ also
  //@ requires backward;
  //@ ensures forward;
  //@ |}
  public void reverse()
}
```

Figure 18: Typestate protocol specification of Figure 17

5.2 Stack Class: Intermixed Specification

In this section, a stack class, which is a common library data structure, is specified using a combination of a typestate protocol and JML. For the stack class, we define three states: `empty`, `hasElement` and `full`. The `empty` state indicates a state where the stack has no elements, represented in the implementation by a `topOfStack` field positioned at `-1`. The `hasElement` state

indicates the stack has one or more elements but less than the maximum capacity of the stack. The `full` state means the container is literally full of elements so no elements can be added any more.

```
public class Stack
{
  //@ [state definition and invariants are omitted]
  //@ invariant \typeof(this.theArray) ==
  \type(java.lang.Object[]);
  //@ invariant theArray.owner == this;
  //@ invariant theArray != null;
  /* spec_public */ private Object [ ] theArray;
  /* spec_public */ private int topOfStack;
  //@ protocol empty -> hasElement;
  //@ protocol hasElement -> hasElement | full;
  //@ ensures theArray[topOfStack] == x;
  //@ ensures topOfStack == \old(topOfStack) + 1;
  public void push( Object x )
}

==>

public class Stack
{
  //@ [state definition and invariants are omitted]
  //@ invariant \typeof(this.theArray) ==
  \type(java.lang.Object[]);
  //@ invariant theArray.owner == this;
  //@ invariant theArray != null;
  /* spec_public */ private Object [ ] theArray;
  /* spec_public */ private int topOfStack;

  //@ {|
  //@ requires empty;
  //@ ensures hasElement;
  //@ ensures theArray[topOfStack] == x;
  //@ ensures topOfStack == \old(topOfStack) + 1;
  //@ also
  //@ requires hasElement;
  //@ ensures hasElement || full;
  //@ ensures theArray[topOfStack] == x;
  //@ ensures topOfStack == \old(topOfStack) + 1;
  //@ |}
  public void push( Object x )
}
```

Figure 19: Typestate protocol specification for Stack Class

In terms of JML specification, the indicator variable `topOfStack` should be decreased as its client removes an element from the stack, and should be increased when an element is added by `push` operation. Since the `topOfStack` can be used for testing the state of a stack in `isFull()` and `isEmpty()`, its value is required to always be equal to the number of stack elements (minus one). We specify the `push()` method using pure JML constructs as well as typestate protocols.

In Figure 19, since the stack is a well-known primitive data type, we omit the state definition and invariants parts as well as other operations such as `pop()` and `top()`. The first four lines declare the class invariants of the stack class. To translate the combination of an ordinary JML specification and a protocol specification with multiple cases, the JML specifications had to be redundantly combined with each typestate specification case.

This appears to highlight a limitation of JML, suggesting potential improvements to its expressive power through a kind of “conjunctive” `also` clause, which would complement the present “disjunctive” `also` clause.

6. CONCLUSION AND FUTURE WORK

In this paper, we extended JML syntax to incorporate tpestate protocol specifications, and presented a corresponding strategy and rules for encoding these tpestate specifications into existing JML constructs. In addition, we proposed a way of handling the violation of code contracts between the behavior of a supertype and subtype. Finally, we showed how our tpestate protocols work through case studies.

With the technique shown in this research, one can more easily use tpestate protocol specifications to specify and verify the behavior of an object-oriented program using JML. Moreover, this research showed how tpestate protocol and pre/post-condition specification can be intermixed so that the developers can specify protocols in a lightweight way, and then naturally extend that specification to describe the full behavior of a component.

In the future, we plan to demonstrate that our technique can be used for specification and verification at a larger scale, since in this paper we only illustrated and verified our technique with simple Java I/O library classes and a primitive data structure class. Also, we plan to implement an integrated tool that supports tpestate protocol specifications based on JML. Finally, in the longer term, we also hope our approach can be integrated with static checkers so that programmers can verify tpestate properties along with other JML assertions at compile time.

7. ACKNOWLEDGMENTS

We thank Gary Leavens for his feedback on earlier drafts of our proposal for adding tpestate specifications to the JML. We would also like to thank Dr. Ko and Dr. Baik for giving helpful advices about the first author's thesis. This work was supported in part by DARPA grant #HR00110710019, NSF grant CCF-0546550, and a Carnegie Mellon University Master of Software Engineering fellowship for the first author.

8. REFERENCES

- [1] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *Technical Report 98-06-rev28*, Iowa State University Department of Computer Science, July 2005.
- [2] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual. Available at <http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/>, retrieved June 2009.
- [3] Jean-Raymond Abrial, Stephen A. Schuman and Bertrand Meyer. A Specification Language. In *On the Construction of Programs*, Cambridge University Press, 1980.
- [4] Kevin Bierhoff and Jonathan Aldrich, Lightweight Object Specification with Tpestates. In *Foundations of Software Engineering*, September 2005.
- [5] Edmund M. Clarke, Jeannette M. Wing, et al., Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, Vol. 28, No. 4, December 1996.
- [6] R. E. Strom and S. Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12:157-171, 1986
- [7] R. DeLine and M. Fähndrich. Tpestates for objects. In *European Conference on Object-Oriented Programming*. Springer-Verlag, 2004.
- [8] C. A. R. Hoare. "An axiomatic basis for computer programming". *Communications of the ACM*, 12(10):576–580, 1969.
- [9] Gary T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *International Conference on Formal Engineering Methods*, pp. 2-34, 2006.
- [10] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In *Foundations of Software Engineering*, 2000.
- [11] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
- [12] R. Duke, G. Rose, and G. Smith. Object-z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
- [13] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.
- [14] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.
- [15] Cheon, Y., Perumendla, A. 2005. Specifying and checking method call sequences in JML. In: Arabnia, H.R., Reza, H. (eds.), Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP'05). vol. II, June 27–29, 2005, Las Vegas, Nevada, CSREA Press, pp. 511–516.
- [16] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API Protocol Checking with Access Permissions. In Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09) (Genova, Italy, July 2009). to appear.