Behavioural Types for Mainstream PLs: On JaTyC et al.

António Ravara
NOVA School of Science and Technology
Lisbon, Portugal

July 27, 2023

0. Plan of the course

Plan of the course

Monday: build a case for type systems

Journey through program verification

Tuesday: type systems 101 and behavioural types

Look at the foundations

Wednesday: behavioural types and PLs

Mungo - memory and thread safe language

Today: Java Typestate Checker et al.

Tutorial presentation, subtyping support, and effort wrt verification

Tomorrow: lab session

An assignment

Plan for today

Yesterday: Mungo

A memory and thread safe language, in detail

JaTyC: applied theory

A tutorial presentation

Details on up/down casting

A bit more of theory

Typestates provide lightweight verification method

Really? Let's test that

Other tools

For Rust and for a Functional Language

1. JaTyC by example

JaTyC: Java Typestate Checker

Statically checks Java code whith typestated objects

Typestates describe the methods available in each protocol state

Guarantees:

- Protocol compliance and completion
- Absence of null-pointer exceptions
- Subclasses' instances respect the protocol of their superclasses

Built on top of the Checker Framework

github.com/jdmota/java-typestate-checker

JaTyC: a re-implementation of Glasgow Mungo

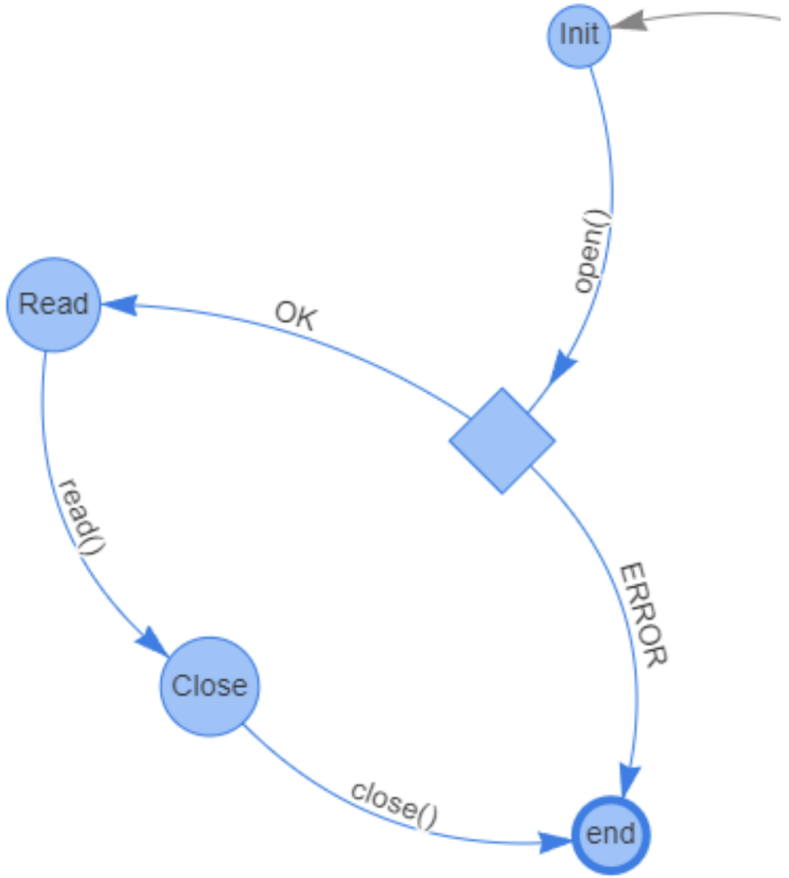
Comparison

Features	Mungo Toolset	Java Typestate Checker	Tests
Basic checking	✓	✓	basic-checking
Decisions on enumeration values	✓	✓	basic-checking
Decisions on boolean values		✓	boolean-decision
`@Requires` `@Ensures`		✓	state-refinement
Nullness checking	🟨 [1]	✓	nullness-checking
Linearity checking	🟨 [2, 3]	✓	linearity-checking, linearity-checking-corner-cases
Force protocol completion	🟨 [4]	✓	protocol-completion, protocol-completion-corner-cases
Class analysis		✓	class-analysis
Protocol definitions for libraries		✓	iterator-attempt1
Droppable objects		✓	droppable-objects

The File example

Protocol compliance

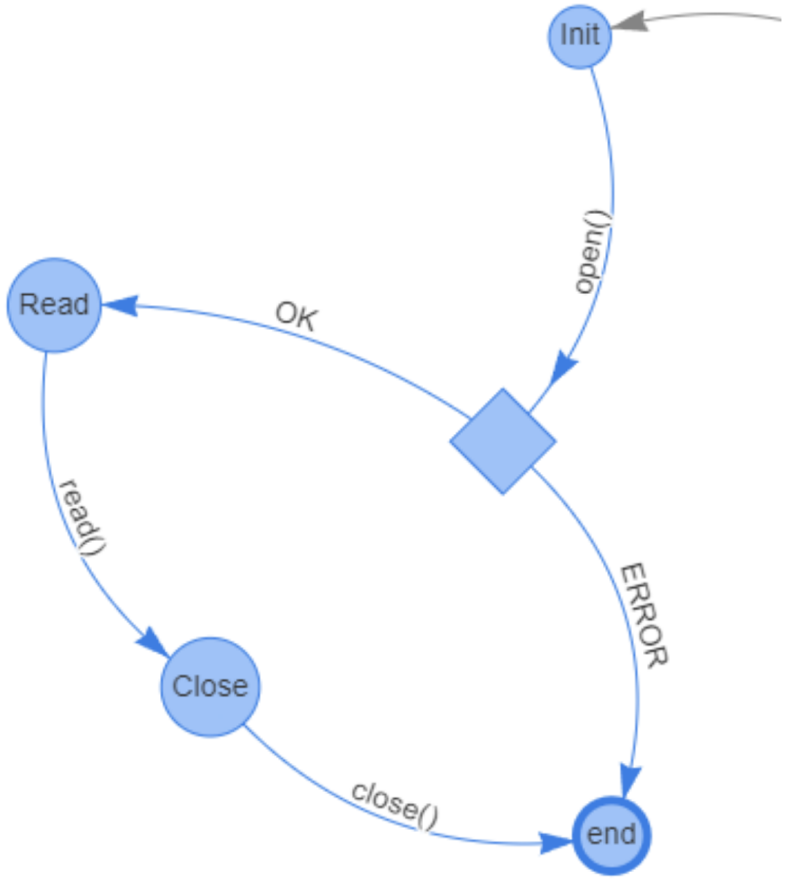
```
File f = new File();

System.out.println(f.read());
// error: Cannot call [read] on State{File, Init}
```



The File example

Protocol compliance

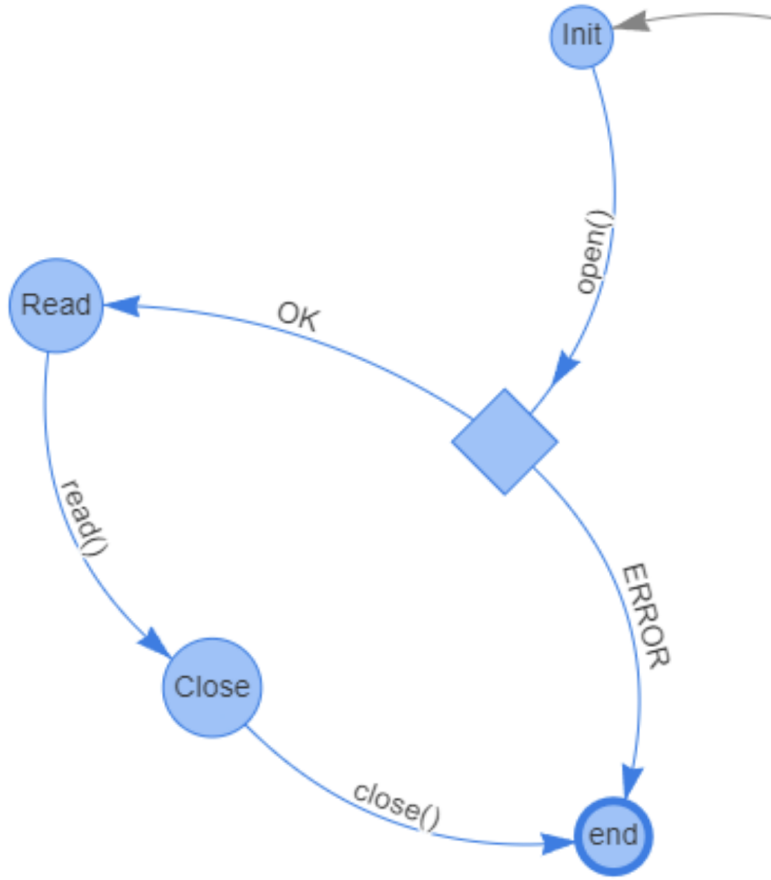
```
File f = new File();

switch (f.open()) {
  case OK:
    System.out.println(f.read());
    break;
  case ERROR:
    break;
}
```



The File example

Protocol completion

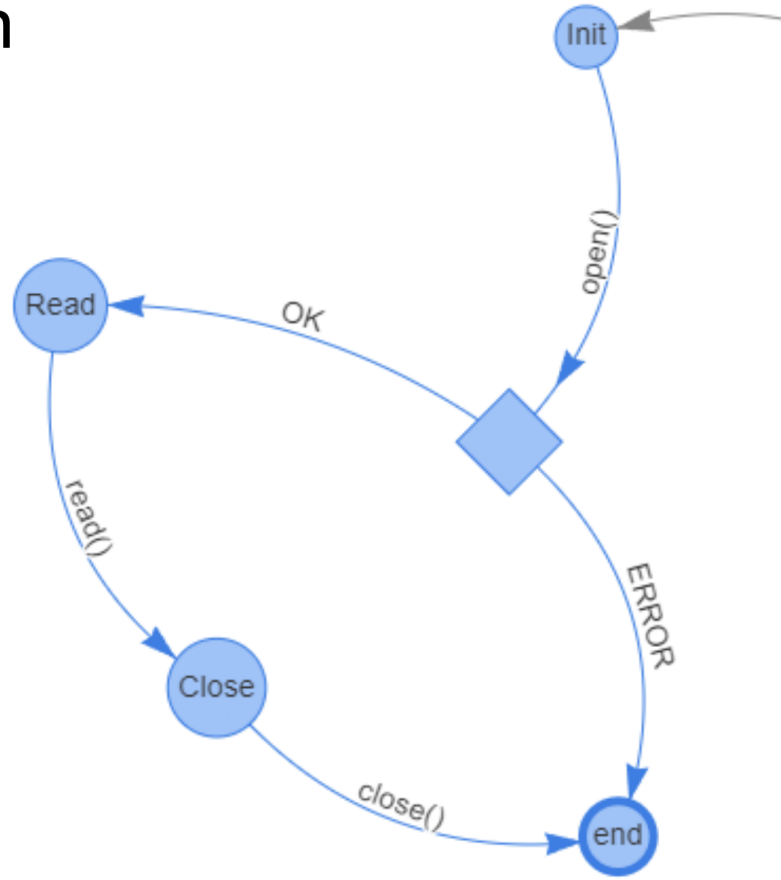
```
File f = new File();

switch (f.open()) {
  case OK:
    System.out.println(f.read());
    break;
  case ERROR:
    break;
}

// error: [f] did not complete its protocol
(found: State{File, Close} | State{File, end})
```



The File example

Protocol compliance & completion

```
File f = new File();

switch (f.open()) {
  case OK:
    System.out.println(f.read());
    f.close();
    break;
  case ERROR:
    break;
}

// OK!
```



The LineReader example

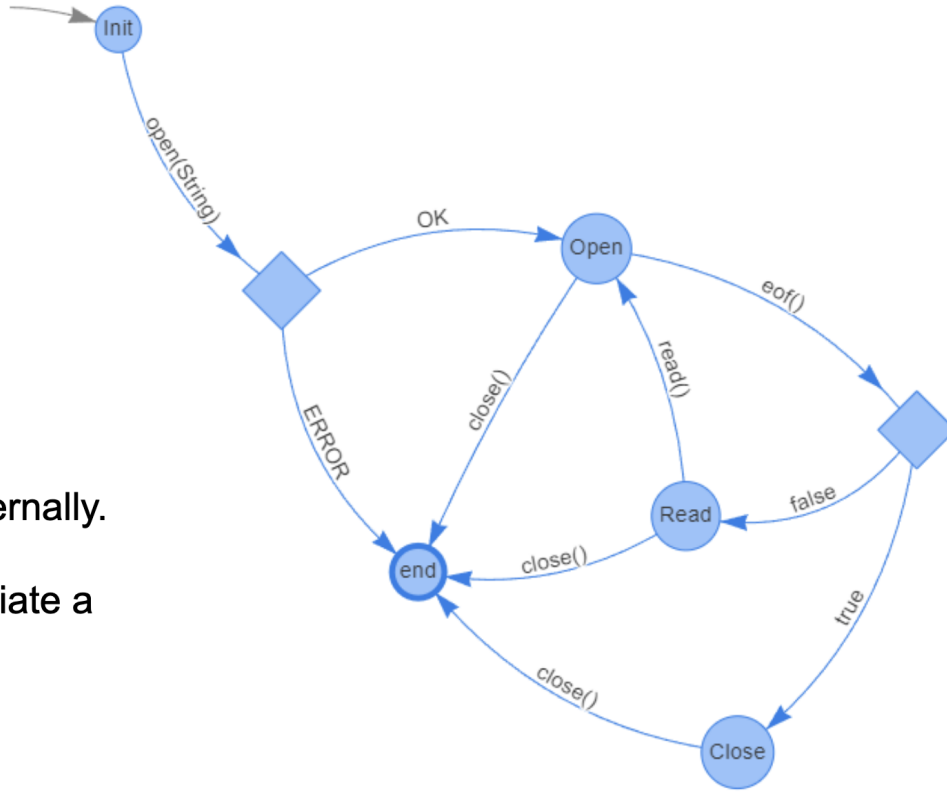
Line Reader

Protocol's happy-path:

`open()` returning `OK`
`read()` while `!eof()`
`close()`

The `LineReader` uses `java.io.FileReader` internally.

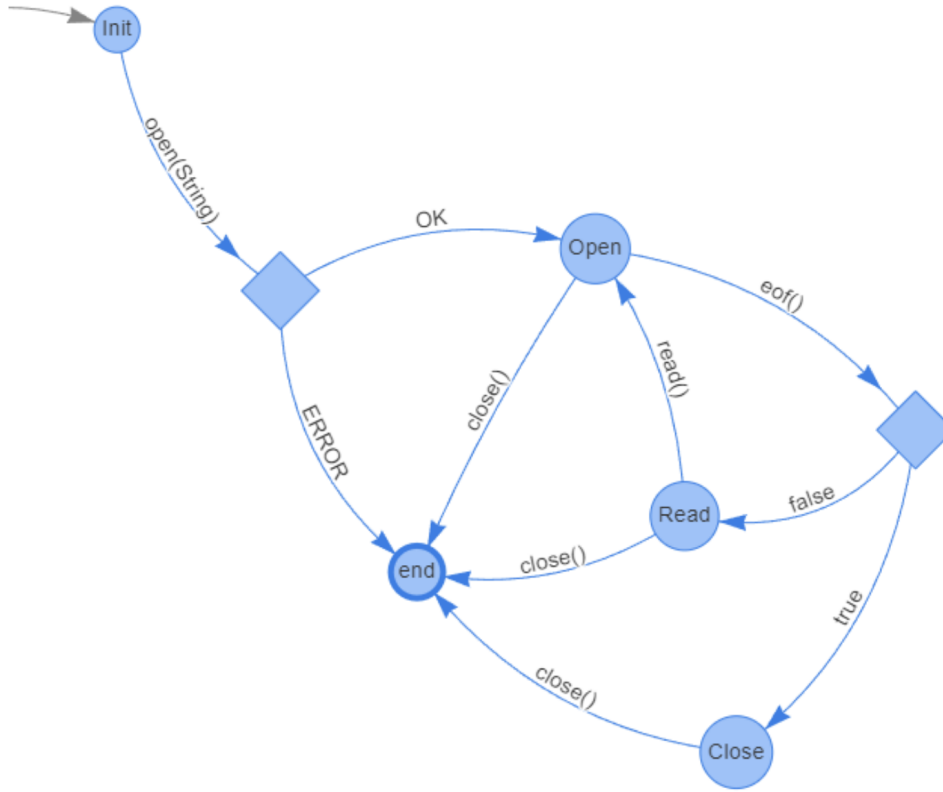
To better check its implementation, we can associate a protocol with `java.io.FileReader` as well!



The LineReader example

Line Reader

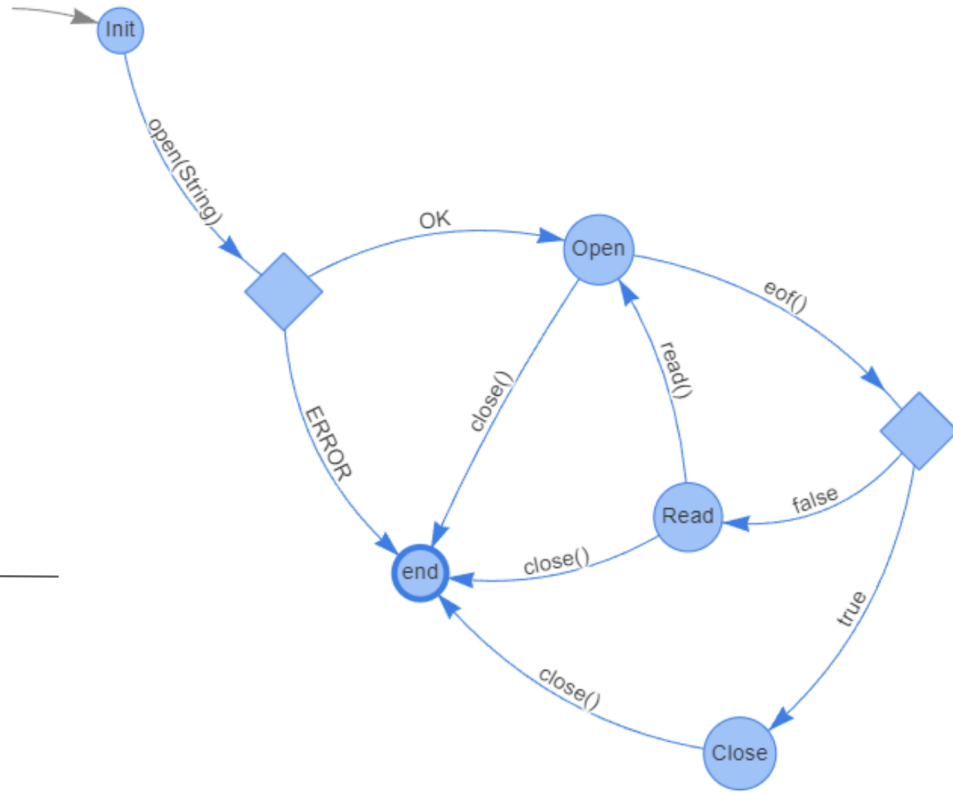
```
@Typestate("LineReader")
public class LineReader {

  private @Nullable FileReader file;
  private int curr;
  public Status open(String filename) {
    /* ... */
    curr = 0; /* ... */
  }
  public String read() {
    /* ... */ curr = file.read(); /* ... */
  }

  public boolean eof() { return curr == -1; }
  public void close() {}
}
```



The LineReader example

Line Reader

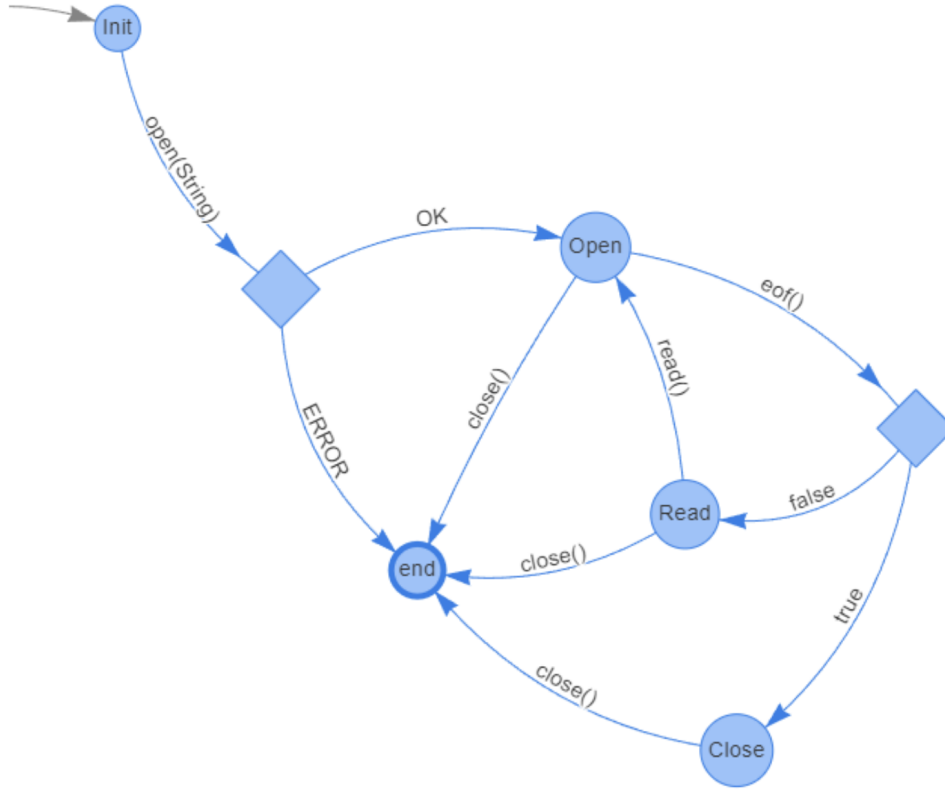
```
@Typestate("LineReader")
public class LineReader {

  private @Nullable FileReader file;
  private int curr;
  public Status open(String filename) {
    /* ... */
    curr = 0; /* ... */
  }
  public String read() {
    /* ... */ curr = file.read(); /* ... */
    // error: Cannot call read on null
  }
  public boolean eof() { return curr == -1; }
  public void close() {}
}
```



The LineReader example

Line Reader

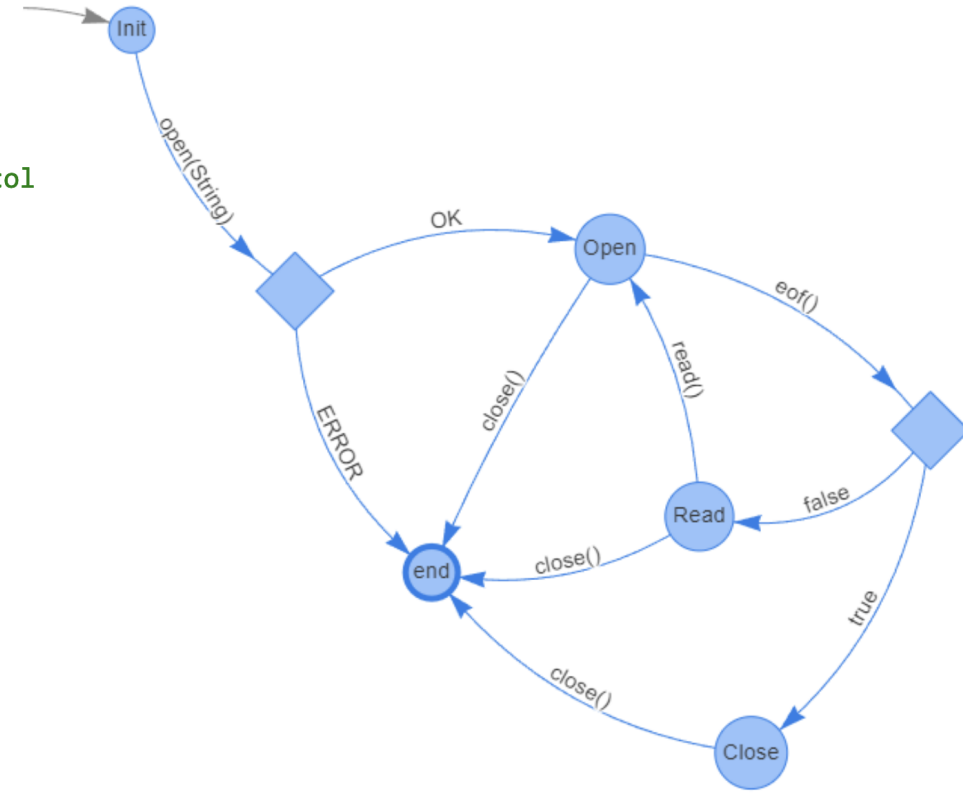
```
@Typestate("LineReader")
public class LineReader {

  private @Nullable FileReader file;
  private int curr;
  public Status open(String filename) {
    /* ... */
    file = new FileReader(filename);
    curr = file.read(); /* ... */
  }
  public String read() {
    /* ... */ curr = file.read(); /* ... */
  }
  public boolean eof() { return curr == -1; }
  public void close() {}
}
```



The LineReader example

Line Reader

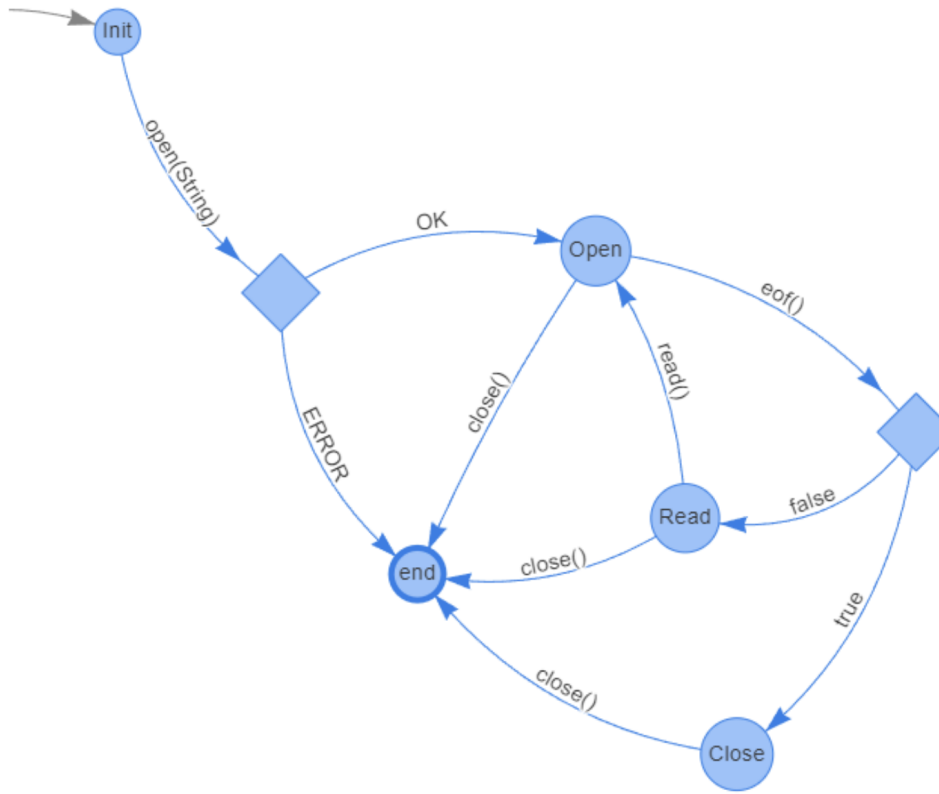
```
@Typestate("LineReader")
public class LineReader {
  // error: [this.file] did not complete its protocol
  private @Nullable FileReader file;
  private int curr;
  public Status open(String filename) {
    /* ... */
    file = new FileReader(filename);
    curr = file.read(); /* ... */
  }
  public String read() {
    /* ... */ curr = file.read(); /* ... */
  }
  public boolean eof() { return curr == -1; }
  public void close() {}
}
```



The LineReader example

Line Reader

```
@Typestate("LineReader")
public class LineReader {

  private @Nullable FileReader file;
  private int curr;
  public Status open(String filename) {
    /* ... */
    file = new FileReader(filename);
    curr = file.read(); /* ... */
  }
  public String read() {
    /* ... */ curr = file.read(); /* ... */
  }
  public boolean eof() { return curr == -1; }
  public void close() { file.close(); }
}
```



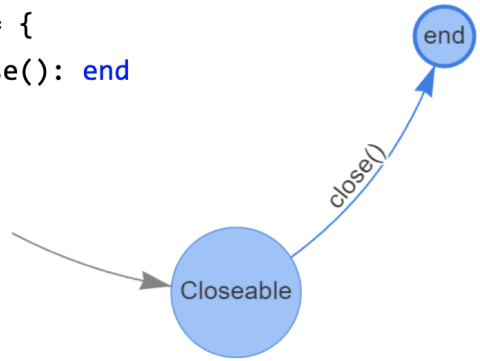
How to deal with libraries?

Protocols for library classes

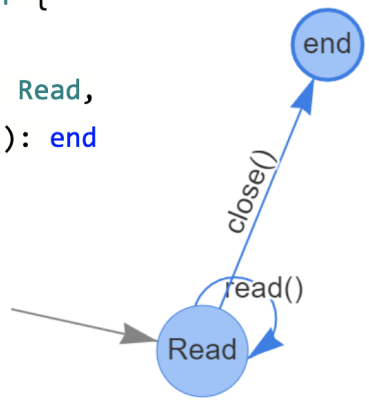
Configuration file

```
java.lang.AutoCloseable=AutoCloseable.protocol
java.io.Reader=Reader.protocol
```

```
typestate AutoCloseable {
  Closeable = {
    void close(): end
  }
}
```




```
typestate Reader {
  Read = {
    int read(): Read,
    void close(): end
  }
}
```

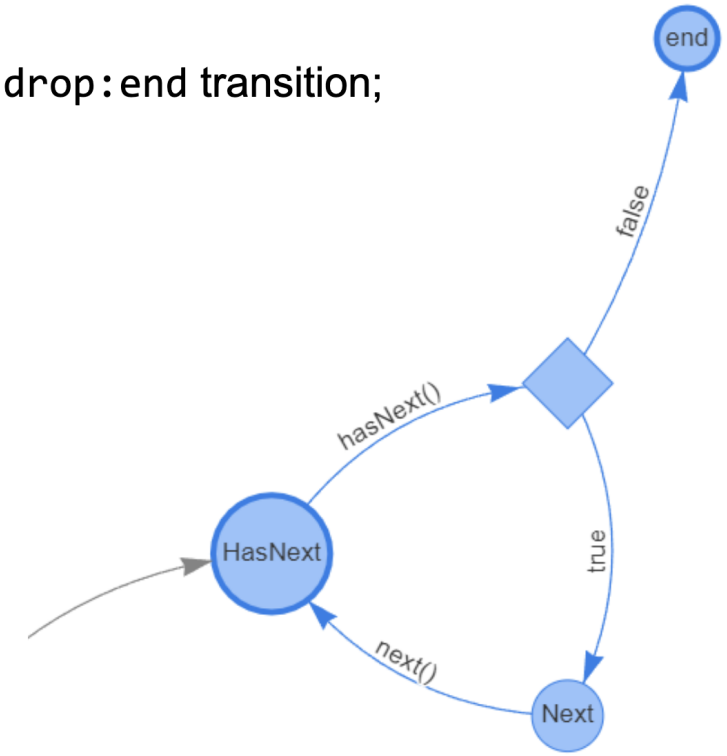


Since `java.io.FileReader` extends `java.io.Reader` (which implements `java.lang.AutoCloseable`), it will inherit the protocol of `java.io.Reader`

Droppable states – not only `end` is final

- One can mark other states as final with the special `drop: end` transition;
- For example, the `HasNext` state is final.

```
typestate Iterator {
  HasNext = {
    boolean hasNext(): <true: Next, false: end>,
    drop: end
  }

  Next = {
    Object next(): HasNext
  }
}
```



A more elaborate example

A Digital Locker application

a workflow of sharing digitally locked files

the owner of the files controls the access to these files

`github.com/Azure-Samples/blockchain`

Roles:

- Owner of the digital asset
- BankAgent – keeper of the digital asset
- ThirdPartyRequestor asks for access to the digital asset

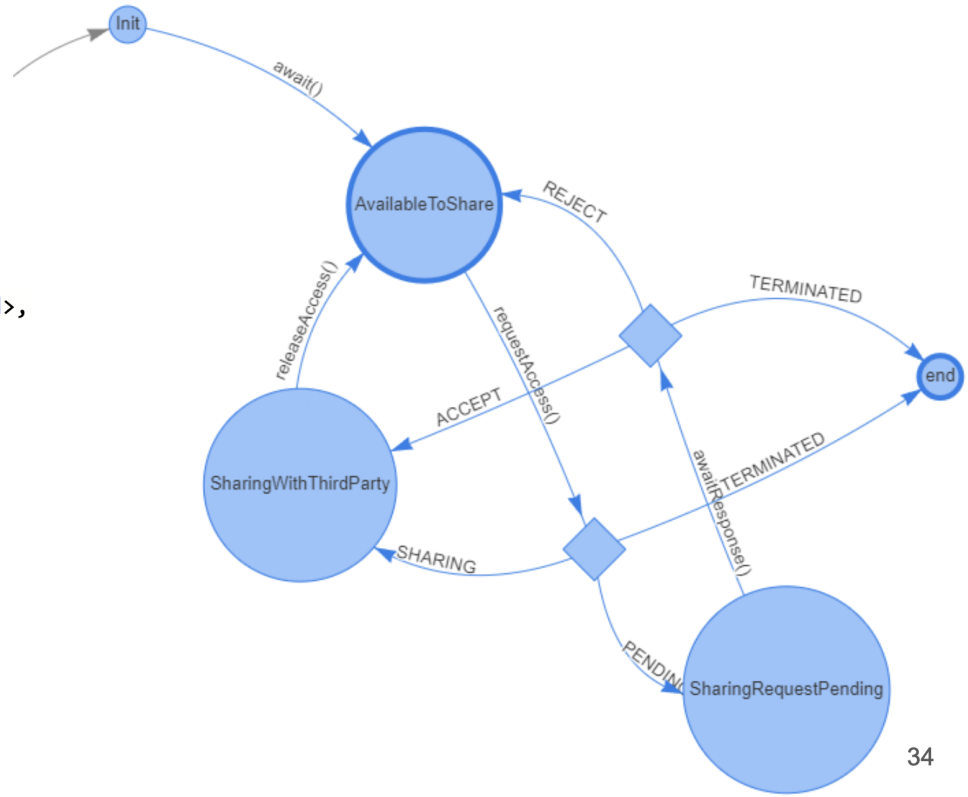
Digital Locker

States:

- Requested: Initial state
- DocumentReview: bank agent reviewed the owner's request
- AvailableToShare: bank agent uploaded the digital asset, which is available for sharing
- SharingRequestPending: owner reviewing a third party's request to access the digital asset
- SharingWithThirdParty: third party accessing the asset

Digital Locker Typestate

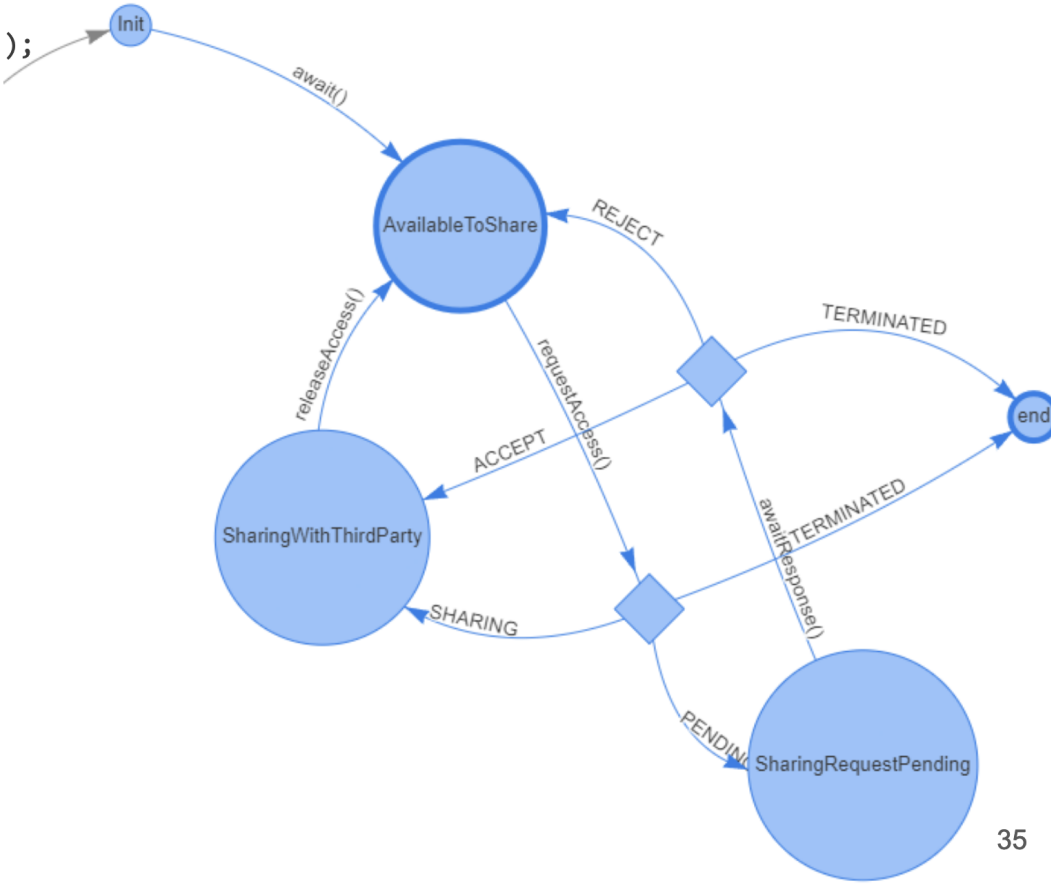
```
typestate ThirdPartyRequestor {
  Init = {
    void await(): AvailableToShare
  }
  AvailableToShare = {
    SharingState requestAccess():
      <PENDING: SharingRequestPending,
       SHARING: SharingWithThirdParty, TERMINATED: end>,
    drop: end
  }
  SharingRequestPending = {
    OwnerResponse awaitResponse():
      <ACCEPT: SharingWithThirdParty,
       REJECT: AvailableToShare, TERMINATED: end>
  }
  SharingWithThirdParty = {
    void releaseAccess(): AvailableToShare
  }
}
```



34

Digital Locker Implementation

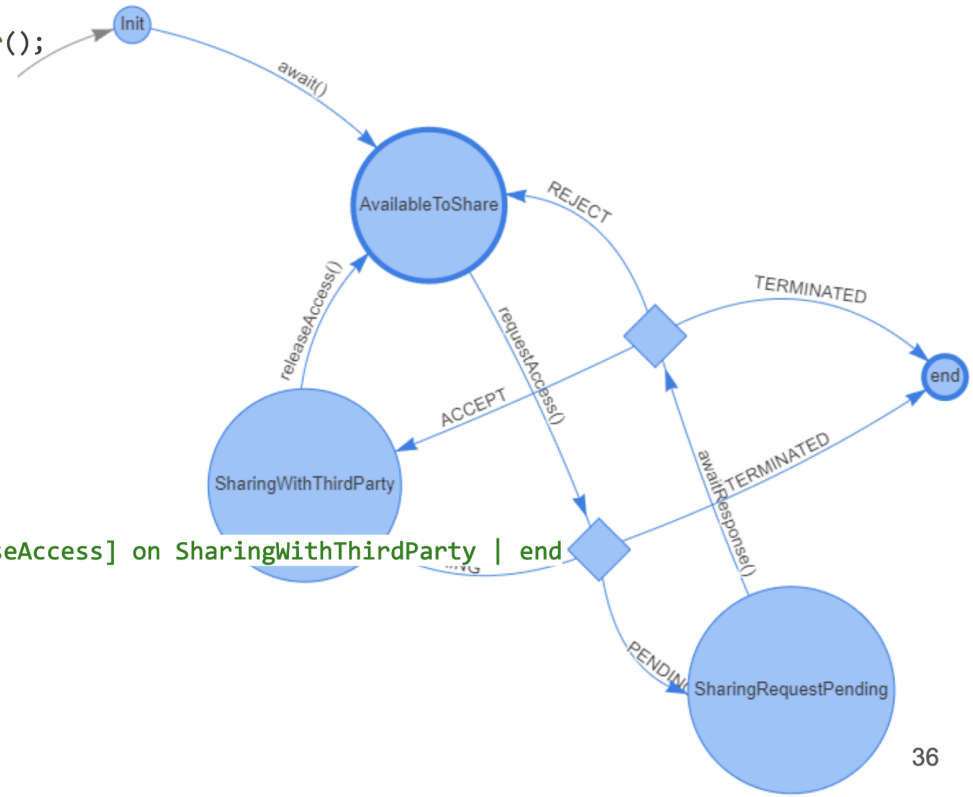
```
ThirdPartyRequestor t = new ThirdPartyRequestor();
t.await();
loop: while (true) {
  switch (t.requestAccess()) {
    case PENDING:
      switch (t.awaitResponse()) {
        case REJECT:
        case TERMINATED:
          break;
        case ACCEPT:
      }
    case SHARING:
      t.releaseAccess();
      break;
    case TERMINATED:
      return;
  }
}
```



35

Digital Locker Implementation

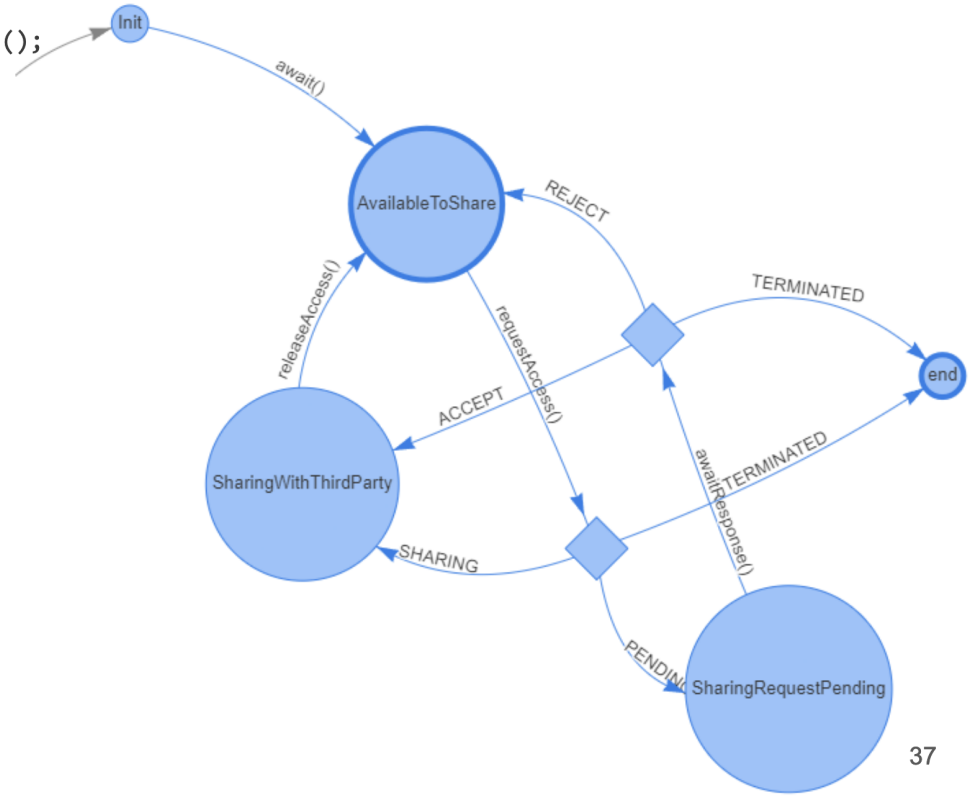
```
ThirdPartyRequestor t = new ThirdPartyRequestor();
t.await();
loop: while (true) {
  switch (t.requestAccess()) {
    case PENDING:
      switch (t.awaitResponse()) {
        case REJECT:
        case TERMINATED:
          break;
        case ACCEPT:
      }
    case SHARING:
      t.releaseAccess(); // Cannot call [releaseAccess] on SharingWithThirdParty | end
      break;
    case TERMINATED:
      return;
  }
}
```



36

Digital Locker Implementation

```
ThirdPartyRequestor t = new ThirdPartyRequestor();
t.await();
loop: while (true) {
  switch (t.requestAccess()) {
    case PENDING:
      switch (t.awaitResponse()) {
        case REJECT:
        case TERMINATED:
          break loop;
        case ACCEPT:
      }
    case SHARING:
      t.releaseAccess(); // OK
      break;
    case TERMINATED:
      return;
  }
}
```

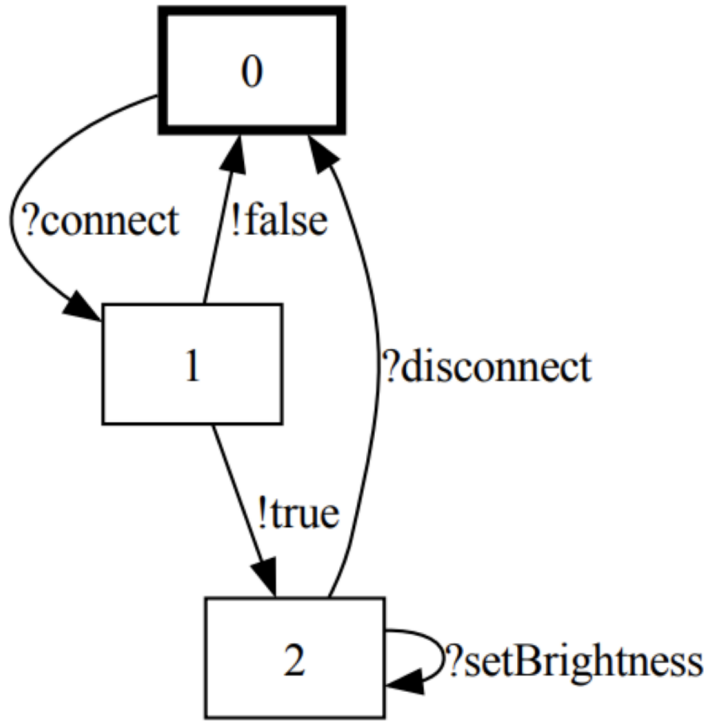


37

Support for subtyping – motivating example

Bulb

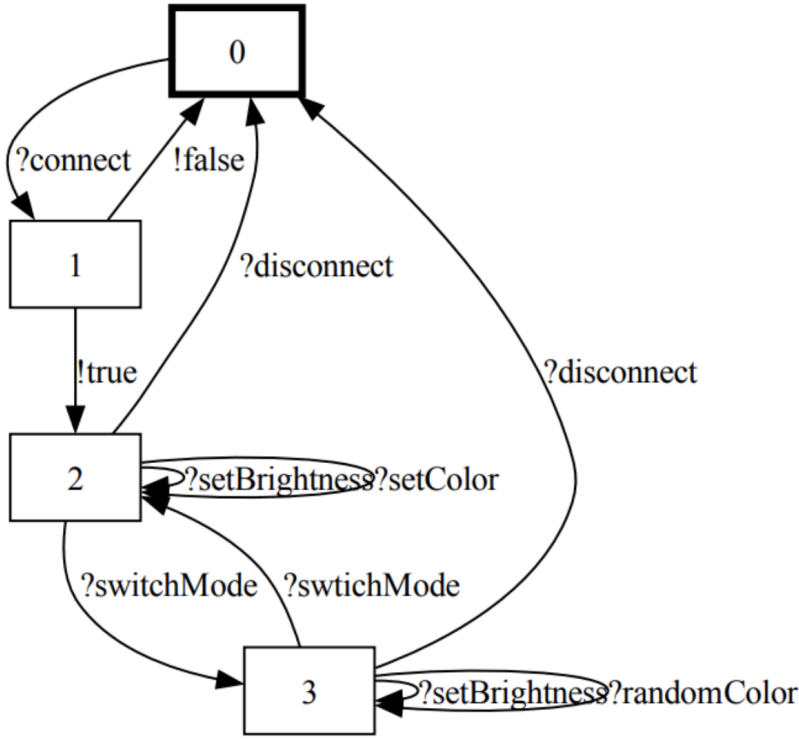
```
typestate Bulb {
  DISCONN = {
    boolean connect(): <true: CONN, false: DISCONN>,
    drop: end
  }

  CONN = {
    void disconnect(): DISCONN,
    void setBrightness(int): CONN
  }
}
```



Support for subtyping – subclass

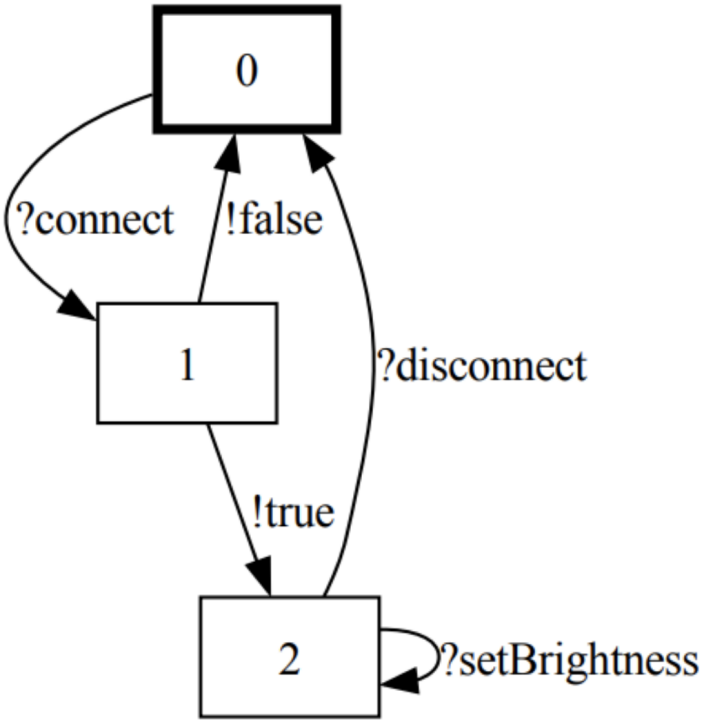
FunnyBulb

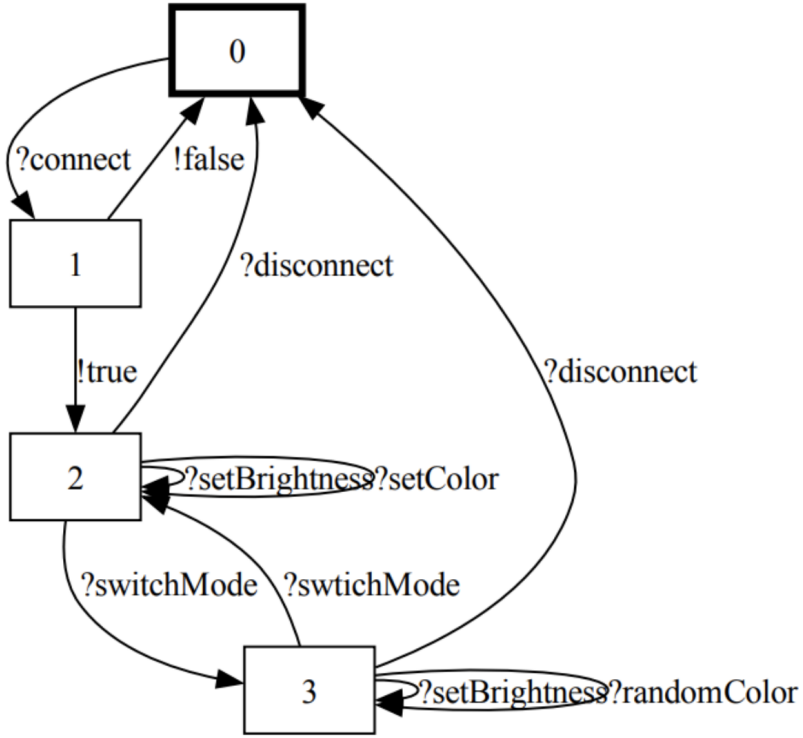
```
typestate FunnyBulb {
  DISCONN = {
    boolean connect(): <true: STD_CONN, false: DISCONN>,
    drop: end
  }
  STD_CONN = {
    void disconnect(): DISCONN,
    void setBrightness(int): STD_CONN,
    Mode switchMode(): <RND: RND_CONN, STD: STD_CONN>,
    void setColor(String): STD_CONN
  }
  RND_CONN = {
    void disconnect(): DISCONN,
    void setBrightness(int): RND_CONN,
    Mode switchMode(): <RND: RND_CONN, STD: STD_CONN>,
    void randomColor(): RND_CONN
  }
}
```



Support for subtyping – extends

FunnyBulb extends Bulb

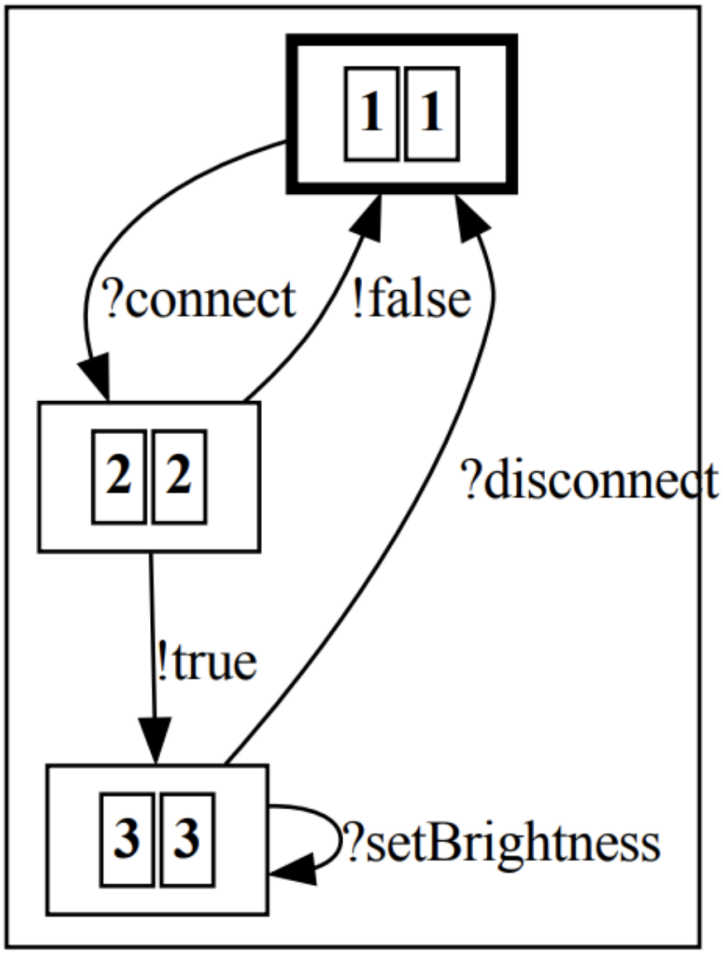




github.com/LBacchiani/session-subtyping-tool

Support for subtyping – extends

FunnyBulb extends Bulb



github.com/LBacchiani/session-subtyping-tool

Polymorphic code

```
import jatyc.lib.Requires;

public class ClientCode {
  public static void example() {
    FunnyBulb f = new FunnyBulb(); // DISCONN
    while (!f.connect()) {} // STD_CONN
    f.switchMode(); // STD_CONN | RND_CONN
    setBrightness(f);
  }

  private static void setBrightness(@Requires("CONN") Bulb b) {
    if (b instanceof FunnyBulb && ((FunnyBulb) b).switchMode() == Mode.RND) {
      ((FunnyBulb) b).randomColor(); // RND_CONN
    }
    b.setBrightness(10); // CONN
    b.disconnect(); // end
  }
}
```

JaTyC: concluding remarks

Limitations / Future work:

- Objects with protocol must be used in a linear way
- No overall support for generics
- No support for dealing with state changes in the presence of exceptions
- No support for multiple inheritance
- No functional verification

0. Plan of the course

1. JaTyC by example

2. Behavioural up/down casting

3. Typestates provide lightweight verification methods

4. Typestates in Rust

5. Freest: a functional language with protocol types

2. Behavioural up/down casting

Motivating example: back to the Car / SUV

Listing 1. Car protocol

```
1  typestate Car {
2    OFF = {
3      boolean turnOn():
4        <true:ON, false:OFF>,
5      drop: end
6    }
7    ON = {
8      void turnOff(): OFF,
9      void setSpeed(int): ON
10   }
11 }
```

Listing 2. SUV protocol (SUV extends Car)

```
1  typestate SUV {
2    OFF = {
3      boolean turnOn():
4          <true:COMF_ON, false:OFF>,
5      drop: end
6    }
7    COMF_ON = {
8      void turnOff(): OFF,
9      void setSpeed(int): COMF_ON,
10     Mode switchMode():
11         <SPORT:SPORT_ON, COMFORT:COMF_ON>,
12     void setEcoDrive(boolean): COMF_ON
13   }
14   SPORT_ON = {
15     void turnOff(): OFF,
16     void setSpeed(int): SPORT_ON,
17     Mode switchMode():
18         <SPORT:SPORT_ON, COMFORT:COMF_ON>,
19     void setFourWheels(boolean): SPORT_ON
20   }
21 }
```

Challenge I: standard simulation approach

Support for casting is essential

```
public static void dispatch(@Requires("ON") Car c) { ··· }
public static void providePoweredSUV(@Requires("OFF") SUV c) {
    if (c.turnOn()) dispatch(c);// Upcast rejected by actual typestate an
}
```

Subtyping from the initial states does not work

```
1  void limitSpeed(@Requires("ON") Car c, int speed) {
2      if (speed > 50) c.setSpeed(50);
3      else c.setSpeed(speed);
4  }
```

SPORT_ON is a subtype of ON

Client code passing to `limitSpeed` an SUV object in typestate SPORT_ON is type-safe

Challenge I: standard simulation approach

Problem: subtyping from (OFF,OFF) does not include (SPORT_ON,ON)

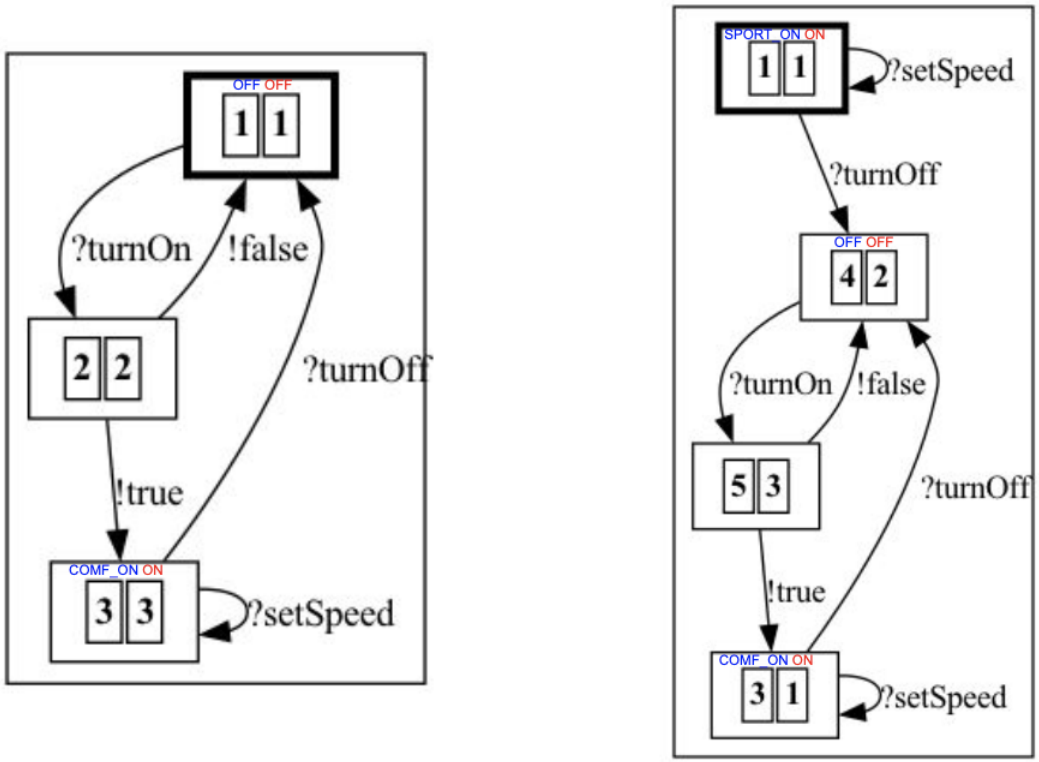


Fig. 1. Simulation relations generated from the subtyping algorithm with the pairs (OFF,OFF) (left) and (SPORT_ON,ON) (right) as input (numbers automatically assigned by a tool).

Solution: run subtyping in all pairs

Challenge II: one-to-many

When upcasting

look for typestates (in the protocol of the target class) that are supertypes of the current one

When downcasting

look for the typestates (again, in the protocol of the target) that are subtypes of the current one

Problem: multiple typestates may be found

Solution:

- upcasting combines the supertypes in an **intersection** type
  method call allowed if permitted by *at least one* element of the intersection
- downcasting combines the supertypes in a **union** type
  method call allowed if permitted by *both* elements of the union

Challenge III: conditional statements

Consider an Electric Car class (`ECar`) which also extends `Car`

```
1  Car c;
2  if (cond) c = new SUV();
3  else c = new ECar();
```

After the if statement, is c an instance of SUV or ECar?

Typestate trees

resemble the class hierarchy

typestate tree – root node `Car`; child nodes `SUV` and `ECar`

Upcast

Let $t ::= t \cup t \mid t \cap t \mid u^{\widetilde{E}} \mid \top \mid \bot$

Theorem 3.8 (Upcast Consistency). *For all t, c and c', we have $t \leq \mathsf{upcast}(t, c, c')$.*

Theorem 3.9 (Upcast Least Upper Bound). *For all t, t', c and c', such that $\mathsf{typestates}(t') \subseteq \mathsf{ProtInputStates}(c')$ and $t \leq t'$, we have $\mathsf{upcast}(t, c, c') \leq t'$.*

Theorem 3.10 (Upcast Preserves Subtyping). *For all t, t', c and c', such that $t \leq t'$, we ha*
$\mathsf{upcast}(t, c, c') \leq \mathsf{upcast}(t', c, c')$.

Downcast

Theorem 3.13 (Downcast Consistency). *For all t, c and c', we have* $\mathsf{downcast}(t, c, c') \leq t$.

Theorem 3.14 (Downcast Greatest Lower Bound). *For all t, c and c', such that* $\mathsf{typestates}(t') \subseteq \mathsf{ProtInputStates}(c')$ *and* $t' \leq t$, *we have* $t' \leq \mathsf{downcast}(t, c, c')$.

Theorem 3.15 (Downcast Preserves Subtyping). *For all t, t', c and c', such that $t \leq t'$, we have* $\mathsf{downcast}(t, c, c') \leq \mathsf{downcast}(t', c, c')$.

Corollary 3.16 (Downcast reverses upcast). *For all t, c and c', we have*

$$t \leq \mathsf{downcast}(\mathsf{upcast}(t, c, c'), c', c).$$

Corollary 3.17 (Upcast reverses downcast). *For all t, c and c', we have*

$$\mathsf{upcast}(\mathsf{downcast}(t, c, c'), c', c) \leq t.$$

Evolve

Method calls change an object state

in which typestates might the object?

$$\texttt{evolve}(\texttt{COMF\_ON} \cup \texttt{SPORT\_ON}, \texttt{switchMode}, \texttt{Mode.SPORT}) = \texttt{SPORT\_ON}$$

Theorem 3.20 (Evolve preserves subtyping). *For all t and t' such that $t \leq t'$, we have that*

$$\texttt{evolve}(t, m, o) \leq \texttt{evolve}(t', m, o)$$

Theorem 3.21 (Evolve and upcast). *For all t, m, o, c and c', we have that*

$$\texttt{upcast}(\texttt{evolve}(t, m, o), c, c') \leq \texttt{evolve}(\texttt{upcast}(t, c, c'), m, o)$$

Theorem 3.22 (Evolve and downcast). *For all t, m, o, c and c', we have that*

$$\texttt{evolve}(\texttt{downcast}(t, c, c'), m, o) \leq \texttt{downcast}(\texttt{evolve}(t, m, o), c, c')$$

Typestate trees

Lift up/downcast and evolve to typestate trees

Similar soundness preservation results hold

In short:

It should be safe to do casting in JaTyC (fingers crossed!)

Try it tomorrow

0. Plan of the course

1. JaTyC by example

2. Behavioural up/down casting

3. Typestates provide lightweight verification methods

4. Typestates in Rust

5. Freest: a functional language with protocol types

3. Typestates provide lightweight verification methods

0. Plan of the course

1. JaTyC by example

2. Behavioural up/down casting

3. Typestates provide lightweight verification methods

4. Typestates in Rust

5. Freest: a functional language with protocol types

4. Typestates in Rust

0. Plan of the course

1. JaTyC by example

2. Behavioural up/down casting

3. Typestates provide lightweight verification methods

4. Typestates in Rust

5. Freest: a functional language with protocol types

5. Freest: a functional language with protocol types

You can't always get what you want

But if you try sometimes you just might find
you get what you need