Stage 1 - Typestate generation

Input example - *File* class

```
class File {

    int linesInFile; int linesRead;
    boolean closed; boolean lineInBuffer; boolean eof;

    //@invariant linesRead >= 0 && linesRead <= linesInFile;
    //@initial linesRead == 0 && linesInFile == 5
            && !closed && !lineInBuffer && !eof;
    void File() { ... }

    ...
}
```

Stage 1 - Typestate generation

Input example - *File* class

```
class File {

        ...

        //@requires linesRead < linesInFile && !closed && lineInBuffer
                    && !eof;
        //@ensures linesRead + 1 <= linesInFile
                    && !closed && !lineInBuffer && !eof;
        string read() { ... }

        ...
}
```

Stage 1 - Typestate generation

Input example - *File* class

```
class File {

        ...

        //@requires linesRead <= linesInFile && !closed && !lineInBuffer
                    && !eof;
        //@ensures (linesRead == linesInFile -> !lineInBuffer && eof)
                    && !closed;
        boolean eof() { ... }

        ...
}
```

Stage 1 - Typestate generation

Input example - *File* class

```
class File {

        ...

        //@requires linesRead == linesInFile && eof && !closed;
        //@ensures linesRead == linesInFile && eof && closed;
        void close() {
                closed = true;
        }
}
```

Stage 1 - Typestate generation

Algorithm steps: initialisation

- Starts by determining the initial state.

Stage 1 - Typestate generation

Algorithm steps: initialisation

- Starts by determining the initial state.
- Determining a state consists on determining the set of methods which the precondition is implied by the constructor's initial condition

Stage 1 - Typestate generation

Algorithm steps: initialisation

- The initial state for the *File* class is:

Stage 1 - Typestate generation

Algorithm steps: initialisation

- The initial state for the *File* class is:
  $S_0 = \{eof\}$

Stage 1 - Typestate generation

Algorithm steps: iteration

- From here, it explores the initial state and every state thereafter

Stage 1 - Typestate generation

Algorithm steps: iteration

- From here, it explores the initial state and every state thereafter
- It manages a queue initialized with the initial state:

Stage 1 - Typestate generation

Algorithm steps: iteration

- From here, it explores the initial state and every state thereafter
- It manages a queue initialized with the initial state:

$$W = \{\{eof\}\}$$

Stage 1 - Typestate generation

Algorithm steps: termination

- While W is not empty, the algorithm takes the state in the head of W, which will be the state to explore

Stage 1 - Typestate generation

Algorithm steps: termination

- While W is not empty, the algorithm takes the state in the head of W, which will be the state to explore

$$W = \{\}$$
$$A = \{eof\}$$

Stage 1 - Typestate generation

Algorithm steps: termination

- While W is not empty, the algorithm takes the state in the head of W, which will be the state to explore

$$W = \{\}$$
$$A = \{eof\}$$

- Before exploring it, the state A will be added to the set of states S, which is the set of states of the typestate

Stage 1 - Typestate generation

Algorithm steps: termination

- While W is not empty, the algorithm takes the state in the head of W, which will be the state to explore

$$W = \{\}$$
$$A = \{eof\}$$

- Before exploring it, the state A will be added to the set of states S, which is the set of states of the typestate

$$S = \{\{eof\}\}$$

Stage 1 - Typestate generation

Algorithm steps

- For each method m in state A, the algorithm will determine the state the typestate transits to when m is executed.

Stage 1 - Typestate generation

Algorithm steps

- For each method *m* in state *A*, the algorithm will determine the state the typestate transits to when *m* is executed.
- Determining the next states is similar to determining the initial state, only it uses the postcondition of *m* instead of the initial condition

Stage 1 - Typestate generation

Algorithm steps

- If *m* is of boolean type and its postcondition specifies two states, two states are determined: One for the true result and other for the false result
- For example, the postcondition of method *eof* implies that:
  - If it returns *true*, there is no more lines to read. This state is valid for the *read* method but not for the *close* method.
  - If it returns *false*, there is at least one more line to read. This state is valid for the *close* method but not for the *read* method.

Stage 1 - Typestate generation

Algorithm steps

- This means that after *eof* there will be two possible states to transit to depending of the returned value
- Its execution causes the typestate to transit into a decision state which will have two transitions, one for each possible result of *eof*:

Stage 1 - Typestate generation

Algorithm steps

- This means that after *eof* there will be two possible states to transit to depending of the returned value
- Its execution causes the typestate to transit into a decision state which will have two transitions, one for each possible result of *eof*:

$$\delta(\{eof\}, eof) = \{eof\_choice\}$$
$$\delta(\{eof\_choice\}, true) = \{close\}$$
$$\delta(\{eof\_choice\}, false) = \{read\}$$

Stage 1 - Typestate generation

Algorithm steps

- States $\{read\}$ and $\{close\}$, since they have not been explored yet, are added to W.

Stage 1 - Typestate generation

Algorithm steps

- The algorithm does the same to every method in A

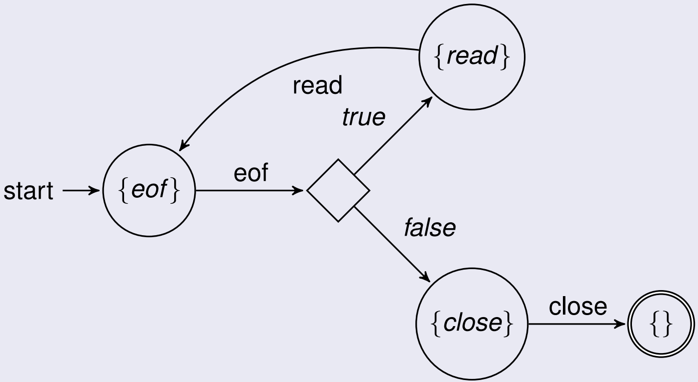
Stage 1 - Typestate generation

Algorithm steps

- The algorithm does the same to every method in A
- After fully exploring the state, the algorithm then explores the state in the head of W and repeats the process until W is empty

Stage 1 - Typestate generation

Output example - Typestate of the *File* class



Stage 2 - Usage generation

Algorithm steps: state id assignment

- The second algorithm starts by creating a set of pairs $(state_{id}, state)$, with $state_{id}$ being the identifier of *state*

Stage 2 - Usage generation

Algorithm steps: state id assignment

- The second algorithm starts by creating a set of pairs $(state_{id}, state)$, with $state_{id}$ being the identifier of *state*
- For the *File* class this set will be: $\{(0, \{eof\}), (1, \{close\})(2, \{read\})\}$

Stage 2 - Usage generation

Algorithm steps: state shared status

- The algorithm also determines the shared status of each state
- A state is considered shared if it only transits to itself or to an equivalent state

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{(0, \{eof\}), (1, \{close\})(2, \{read\})\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\delta(\{close\}, close) = \{\}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{\underline{(0, \{eof\})}, (1, \{close\})(2, \{read\})\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\delta(\{close\}, close) = \{\}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
      0 = _{eof; _}
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{\underline{(0, \{eof\})}, (1, \{close\})(2, \{read\})\}$

$\boldsymbol{\delta(\{eof\}, eof) = \{eof\_choice\}}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\delta(\{close\}, close) = \{\}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
        0 = _{eof; <_ + _>}
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{\underline{(0, \{eof\})}, \mathbf{(1, \{\mathit{close}\})}, \mathbf{(2, \{\mathit{read}\})}\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\boldsymbol{\delta(\{eof\_choice\}, true) = \{close\}}$

$\boldsymbol{\delta(\{eof\_choice\}, false) = \{read\}}$

$\delta(\{close\}, close) = \{\}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
        0 = lin{eof; <1 + 2>}
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{(0, \{eof\}), \underline{(1, \{close\})}, (2, \{read\})\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\delta(\{close\}, close) = \{\}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
      0 = lin{eof; <1 + 2>}
      1 = _{close; _}
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{(0, \{eof\}), \underline{(1, \{close\})}, (2, \{read\})\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\boldsymbol{\delta(\{close\}, close) = \{\}}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
        0 = lin{eof; <1 + 2>}
        1 = lin{close; end}
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{(0, \{eof\}), (1, \{close\}), \underline{(2, \{read\})}\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\delta(\{close\}, close) = \{\}$

$\delta(\{read\}, read) = \{eof\}$

```
usage lin{File; 0} where
        0 = lin{eof; <1 + 2>}
        1 = lin{close; end}
        2 = _{read; _}
```

Stage 2 - Usage generation

Algorithm steps: state translation

- Using the previous set and the transition relation of the typestate, each state is then translated into an usage state

$\{(\mathbf{0}, \{\boldsymbol{eof}\}), (1, \{close\}), \underline{(2, \{read\})}\}$

$\delta(\{eof\}, eof) = \{eof\_choice\}$

$\delta(\{eof\_choice\}, true) = \{close\}$

$\delta(\{eof\_choice\}, false) = \{read\}$

$\delta(\{close\}, close) = \{\}$

$\boldsymbol{\delta(\{read\}, read) = \{eof\}}$

```
usage lin{File; 0} where
        0 = lin{eof; <1 + 2>}
        1 = lin{close; end}
        2 = lin{read; 0}
```

Stage 2 - Usage generation

Output example - Usage of the *File* class

```
usage lin{File; Q1} where
    0 = lin{eof; <1 + 2>}
    1 = lin{close; end}
    2 = lin{read; 0}
```

Stage 3 - Object usage state inference

Algorithm steps - overview

- Goes through all the methods of each class (following the order of the usage) and, for each one:

Stage 3 - Object usage state inference

Algorithm steps - overview

- Goes through all the methods of each class (following the order of the usage) and, for each one:
  1. Determines the usage state of every parameter using the precondition of the method

Stage 3 - Object usage state inference

Algorithm steps - overview

- Goes through all the methods of each class (following the order of the usage) and, for each one:
  1. Determines the usage state of every parameter using the precondition of the method
  2. Determines the usage state of the return type using the postcondition of the method

Stage 3 - Object usage state inference

Algorithm steps - overview

- Goes through all the methods of each class (following the order of the usage) and, for each one:
  1. Determines the usage state of every parameter using the precondition of the method
  2. Determines the usage state of the return type using the postcondition of the method
  3. Analyses the code of the method and:

Stage 3 - Object usage state inference

Algorithm steps - overview

- Goes through all the methods of each class (following the order of the usage) and, for each one:
  1. Determines the usage state of every parameter using the precondition of the method
  2. Determines the usage state of the return type using the postcondition of the method
  3. Analyses the code of the method and:
     - For every initialization, sets the usage state of the initialized variable with the usage state of the value

Stage 3 - Object usage state inference

Algorithm steps - overview

- Goes through all the methods of each class (following the order of the usage) and, for each one:
  1. Determines the usage state of every parameter using the precondition of the method
  2. Determines the usage state of the return type using the postcondition of the method
  3. Analyses the code of the method and:
     - For every initialization, sets the usage state of the initialized variable with the usage state of the value
     - For every call, changes the current usage state of the object the method was called

Stage 3 - Object usage state inference

Algorithm steps - determining the usage state of an object

- When determining the usage state of an object, the algorithm checks the first usage state that has a set of method whose preconditions are implied by the assertions that specifies its state