On using VeriFast, VerCors, Plural, and KeY to check object usage

João Mota, Marco Giunti, António Ravara

NOVA School of Science and Technology, Portugal

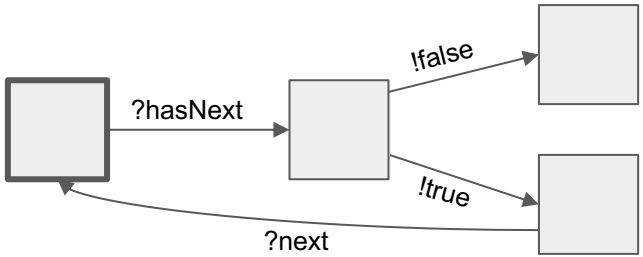
ECOOP 2023

Background

- Static analysis for objected-oriented programming (*OOP*)
- Underlying idea:
  - Objects have **internal state** that determines which operations are “safe”
  - E.g. a **stack** object in **empty** state does not support **pop** operation
  - Set of available operations changes with time, e.g. method *n* is not available until after method *m* has been called
- This approach has been pursued under several names, e.g.
  - Typestates
  - Non-uniform objects

Typestates

- **Typestates** define object's behavior in terms of a *state machine: a method execution* is described as a *state transition*
- A typestate associated to a *class* can be seen as the **protocol** that instances must follow

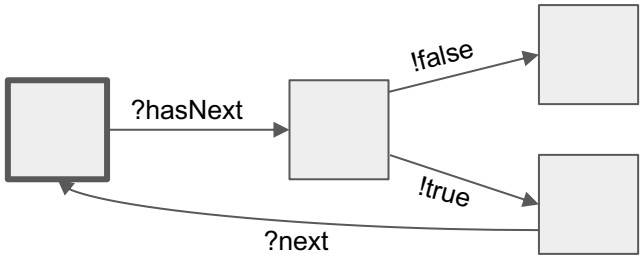



```
while (iterator.hasNext()) {
    Object value = iterator.next();
}
```

Safety of protocols = compliance + completion

Well-typed programs inherit the following protocol properties:

- *Compliance:* methods called in correct order
  - E.g. `x.next()` always “encapsulated” in context `if x.hasNext(){e1; ❍; e2;} else e3` where `e1` does not change the state of `x`
- *Completion:* when the program terminates, all protocols have “ended”




```
while (iterator.hasNext()) {
    Object value = iterator.next();
}
```

Main limitation: sharing not supported

- Static analysis based on typestates or uniform objects typically relies on ownership control: objects' behaviour is **strictly linear**
- Main limitation: objects **cannot be** stored in **shared** data structures
- Typical **unhandled** scenarios**:** a **file reader**; a **collection** of file readers; an **iterator**
- Several less extreme techniques for **control of aliasing** have been studied, but can we use them while relying on **typestates**?

Research challenge

- We are interested in (re-)**using** static analysis **tools** and techniques for **OOP**
- Requirements -- the program analysis must:
  - Be instrumented to use **typestate-**based reasoning in order to ensure protocol **compliance** and **completion**
  - Support **sharing**
- **This talk:**
  - We **assess** whether this can be done in **four** mature **Java tools**

Outline

- Research methodology
- Use cases:
  - File reader
  - Singly-linked list with double handle (head and tail)
  - Iterator
- Experiments
- Assessment

Methodology: Research Question (*RQ*)

Are current static verification tools capable of verifying:

- Protocol **compliance**
- Protocol **completion**

even when objects are **shared** in collections?

Methodology: state-of-the-art Java tools

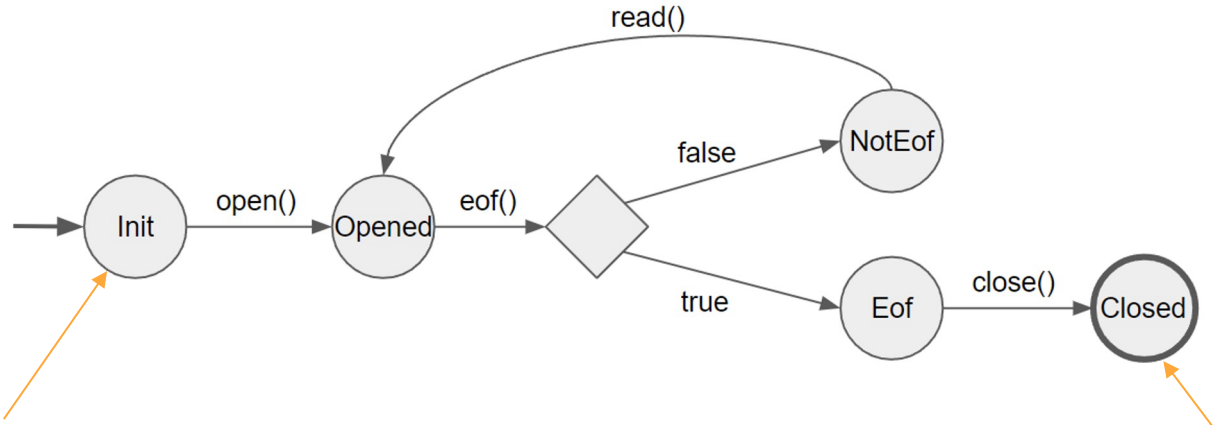
We report our experience on using four tools for Java:

	Description	**Specifications / Underlying logic**	**Interactivity**	**Active**
VeriFast	Modular verifier for C and Java	Separation logic + Fractional permissions	IDE	Yes
VerCors	Modular verifier for C, Java, OpenCL and PVL	Permission-based concurrent separation logic + Fractional permissions	None	Yes
KeY	Verifier for sequential Java programs	JML + First-order dynamic logic	Interactive theorem prover	Yes
Plural	Eclipse plugin to verify Java code	Typestates + Access permissions	None	No

Methodology: RQ assessment

- *RQ: Can tools ensure **protocol compliance** and **completion**, even with objects **shared in collections**?*
- Identify elaborated **use cases** and stress the tools in order to positively answer to the RQ
- **Evaluate** the programmer's effort

Use cases: File Reader's protocol

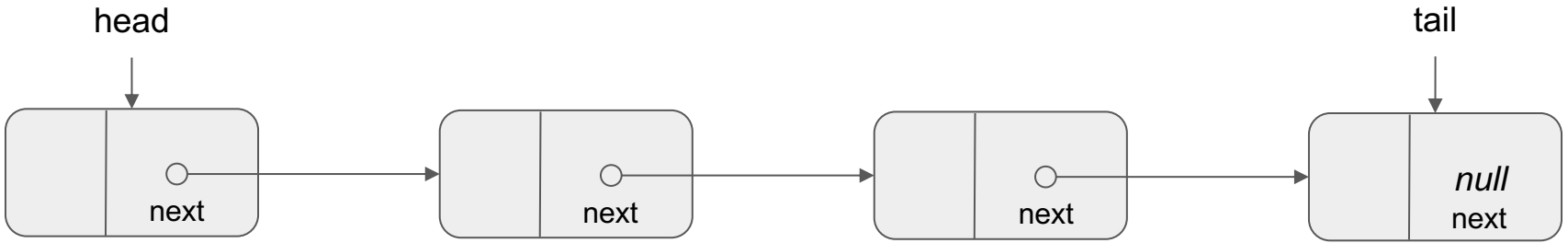


Initial_state
= Init

```
FileReader file = new FileReader();
file.open();
while (!file.eof()) {
  file.read();
}
file.close();
```

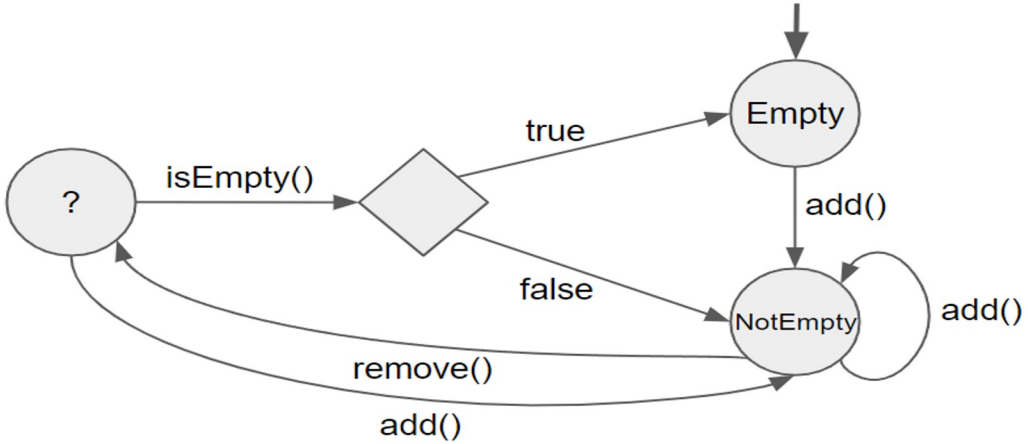
Closed ∈
Ended_states

Use cases: Linked-list with double handle (*2-Linked-list*)



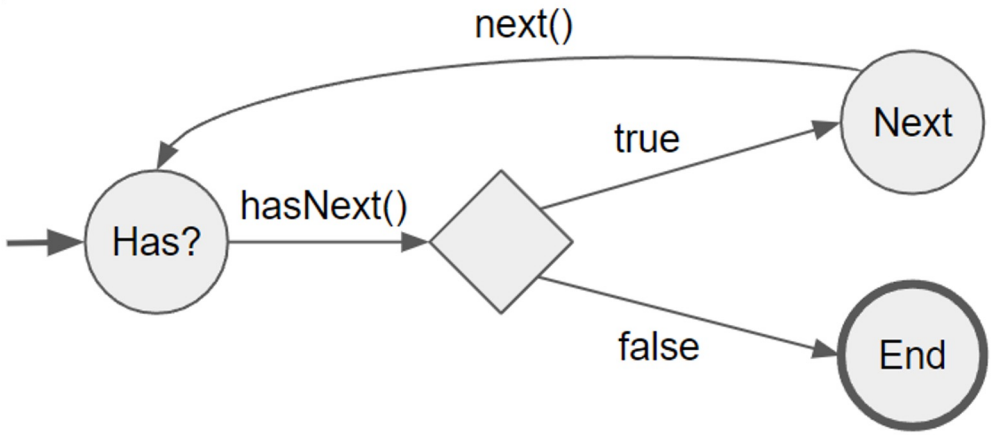
Singly-linked list with double-handle: list has head and tail fields

- Tail field efficient to add entries; structure useful e.g. to implement queues
- Challenging for interplay between *ownership and aliasing:* next and tail might be aliases



Use cases: Iterator

```
public class Iterator {
  private Node curr;
  public Iterator(Node head) {
    curr = head;
  }
  public boolean hasNext() {
    return curr != null;
  }
  public Object next()  {
    Object value = curr.value;
    curr = curr.next;
    return value;
  }
}
```




```
while (iterator.hasNext()) {
    Object value = iterator.next();
}
```

Usage example: consuming a list of file readers

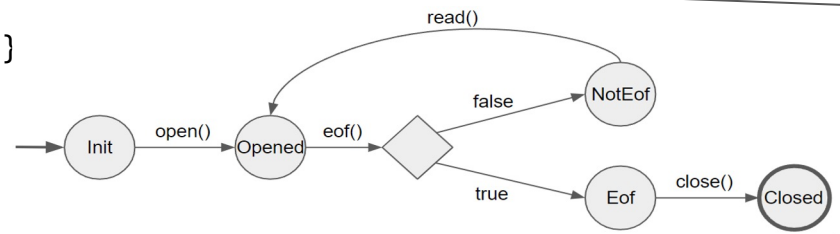
```
static void main() {
  LinkedList list = new LinkedList();
  FileReader f1 = new FileReader("a.txt");
  FileReader f2 = new FileReader("b.txt");
  FileReader f3 = new FileReader("c.txt");
  list.add(f1);
  list.add(f2);
  list.add(f3);

  useFiles(list);
}
```

Pre-condition: all files in the list in the ***Init*** state

```
static void useFiles(LinkedList list){
  LinkedListIterator it = list.iterator();
  while (it.hasNext()) {
    FileReader f = it.next();
    f.open();
    while (!f.eof()) f.read();
    f.close();
  }
}
```

Post-condition: all files in the list in the ***closed*** state



Experiments: encoding FileReader's protocol

Protocol **states** are encoded with:

- A runtime *state* **field** (VeriFast)
- A **ghost** *state* **field** (VerCors, KeY)
- **Typestates** (Plural)

```
private int state; //VeriFast

//@ ghost private int state;  //VerCors

/*@ ghost private spec_public int state;*/ //Key

@Refine({
  @States(value={"init", "opened", "closed"}, refined="alive"),
  @States(value={"eof", "notEof"}, refined="opened")
}) //Plural
```

Protocol **steps** are encoded with **method annotations** in all tools.

```
//@ requires filereader(this, STATE_INIT, _); //VeriFast
//@ ensures filereader(this, STATE_OPENED, _); //VeriFast

//@ requires state == 1; //VerCors
//@ ensures state == 2;  //VerCors

/*@ requires state == STATE_INIT;
    ensures state == STATE_OPENED; @*/ //Key

@Unique(requires="init", ensures="opened") //Plural
```

Experiments: encoding 2-Linked-list & Iterator's protocols

VeriFast, VerCors, KeY

We represent the structure (**memory footprint**) of the list and iterator using:

- **Separation logic** (VeriFast, VerCors)
- First-order **dynamic logic** (KeY)

We reason about the state of the **2-linked-list** with a **ghost** sequence of **values**.

We reason about the state of the **iterator** with:

- A **current** node **field**
- A **ghost sequence** of values **seen**
- A **ghost sequence** of values **to see**

Plural

- **Challenge:** how to represent the memory footprint in a model with only typestates and access permissions?

- **Encoding**
  - The list has **exclusive access** (via *unique*) to the first node
  - Each node has exclusive access to the next node
  - We use *unique* to enforce that the list is not circular

  - **Limitation:** as in Rust, we cannot implement a 2-linked-list using both a head and a tail (mutable fields next and tail might be aliases)

  - We can't "transfer" the permission (cf. Sep. logic)

RQ Assessment: protocol compliance and completion

Compliance is assessed by:

- Verify that only the allowed methods in each state can be called
- VeriFast, VerCors, and KeY:
  - By means of pre-conditions and post-conditions
- Plural
  - By means of typestate abstraction directly supported

Completion is assessed by:

- Checking that if and when the program terminates, all the objects are in the *ended* states of their protocols

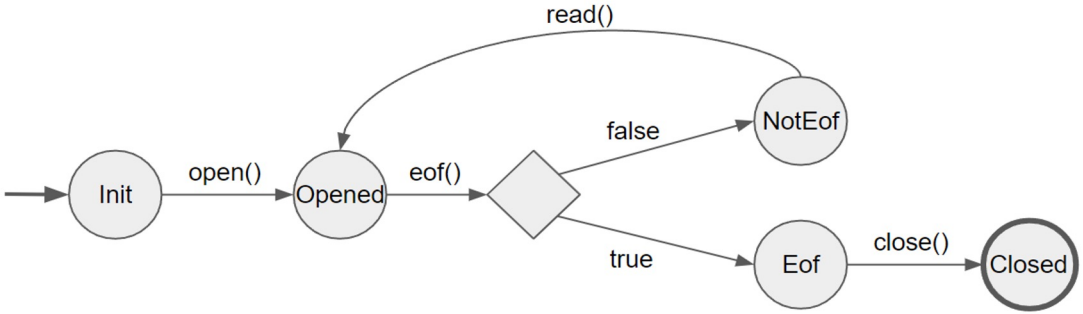
RQ Assessment: use cases' compliance

Can we implement solutions to compliance for the the use cases?

- **File reader**
  - Successfully implemented in all tools

- **2-Linked-list and iterator**
  - Successfully implemented in VeriFast, VerCors, and KeY:
    - VeriFast and VerCors: *deductive reasoning* was often required
    - KeY: often results required *interactivity*
    - In all three: there was *specification burden*

Example: detection of non-compliance in Plural




```
FileReader f = new FileReader();
f.open();
while (f.eof()) { // bug! The guard should be !f.eof()
  f.read(); // expected error: argument this must be in state [notEof] but is in [eof]
}
f.close(); // expected error: argument this must be in state [eof] but is in [notEof]
```

Plural successfully detects when protocol compliance does not hold

RQ Assessment: use cases' completion

Can we implement solutions to completion for the use cases?

With workarounds, we verify protocol completion in

VeriFast, **VerCors**, and **KeY**:

- VeriFast and VerCors: with *ghost counters* counting active file readers
- KeY: with *universal quantification*

We could not verify completion in **Plural**

- permissions may be "dropped" and we cannot overcome the issue as we did in the other tools by relying on deductive reasoning (e.g. to count the number of active objects)

Example: useFiles completion in VeriFast

```
public static void useFiles(LinkedList list)
  //@ requires list != null &*& llist(list, _, _, ?l) &*& tracker(length(l)) &*& foreachp(l, INV(FileReader.STATE_INIT));
  //@ ensures list != null &*& llist(list, _, _, l) &*& tracker(0) &*& foreachp(l, INV(FileReader.STATE_CLOSED));
{
  LinkedListIterator it = list.iterator();
  while (it.hasNext())
    /*@ invariant
        it != null &*& iterator(list, it, _, l, ?a, ?b) &*& tracker(length(b)) &*&
        foreachp(a, INV(FileReader.STATE_CLOSED)) &*&
        foreachp(b, INV(FileReader.STATE_INIT));
     @*/
  {
    FileReader f = it.next();
    //@ open foreachp(b, _);
    //@ open INV(FileReader.STATE_INIT)(f);
    f.open();
    while (!f.eof()) //@ invariant f != null &*& filereader(f, FileReader.STATE_OPENED, _);
    { //@ open filereader(f, _, _);
      f.read();
    }
    f.close();
    //@ close foreachp(nil, INV(FileReader.STATE_CLOSED));
    //@ close foreachp(cons(f, nil), INV(FileReader.STATE_CLOSED));
    //@ foreachp_append(a, cons(f, nil));
  }
  //@ dispose_iterator(it);
}
```

Receive a list with file readers in the *Init* state and finish their protocol

counter at 0

All seen files are **closed**, the others are at the **init** state

Return the permission to the list the iterator got

RQ assessment: summary

*RQ: Are current static verification tools capable of verifying protocol **compliance** and **completion** even when objects are **shared in collections**?*

	Protocol compliance	**Protocol completion**
VeriFast	Yes, but with effort	Yes, but with workarounds
VerCors	Yes, but with effort	Yes, but with workarounds
KeY	Yes, but with effort	Yes, but with workarounds
Plural	Yes for the file reader, with no effort. We could not implement a 2-Linked-list	No, but could be done by allowing only permissions for *ended* objects to be “dropped”

Discussion

- In state-of-the-art OOP tools, **protocols** are **not first-class entities:** they are usually encoded with method contracts
- **First-class protocols** would make reasoning on relevant properties **more easy** than on lower level encodings
- Our previous experience on **typestate-checking** Java programs in the presence of **inheritance*** (but without sharing) confirms this intuition

(*) Bacchiani et al. *A Java typestate checker supporting inheritance*. Sci. Comput. Program. 221 (2022)

Discussion

We motivate the need for **lightweight methods** to statically guarantee:

- protocol **compliance**
- protocol **completion**

in the presence of several patterns of **sharing**

(like objects with protocol stored in collections)

while supporting OOP features as **inheritance**, **upcast/downcast**, **generics**, …

Thank you!

(running examples: github.com/jdmota/tools-examples)