On using Plural to check object usage

João Mota Marco Giunti
António Ravara
NOVA LINCS and NOVA School of Science and Technology, Portugal

May 6, 2022

Contents

1 Introduction

The goal of this report is twofold: (1) assess if Plural [3] can check the **correct use of objects with protocols**, including **protocol completion**, even with objects **shared in collections**; (2) evaluate the **programmer's effort** in making the code acceptable to the tool.

When programming, one naturally defines objects where their method's availability depends on their state [5, 1]. One might represent their intended usage protocol with an automaton or a state machine [7, 6, 4]. **Behavioral types** allow us to statically check if all code of a program respects the protocol of each object. In session types approaches, objects associated with protocols are usually forced to be used in a linear way to avoid race conditions, which reduces concurrency and restricts what a programmer can do. Given that sharing of objects is very common, it should be supported. For example, pointer-based data structures, such as linked-lists, used to implement collections, usually rely on internal sharing. Such collections may also be used to store objects with protocols and state which needs to be tracked. Moreover, it is crucial that all protocols complete to ensure necessary method calls are not forgotten and resources are freed.

Even though Plural does not seem to be maintained any longer, we were able to install it by downloading its source[1], and the source code of its dependencies, and installing them in Eclipse Juno (an old version from 2012).

We conclude that Plural has the richest set of access permissions we know of, allowing even the sharing of mutable data thanks to state guarantees. Nonetheless, we find some drawbacks:

- The lack of support for predicates (to define recursive properties) makes it difficult (if not impossible) to model more complex data structures, such as linked-lists, which have aliasing between the *next* field of the second to last node and the *tail* field;

[1] `https://code.google.com/archive/p/pluralism/`

- Although there is support for parametric fractional permissions, there seems to be no support for parametric typestates, which would allow one to model a list with objects in different states that evolve;
- Thread-local shared references require locking to be modified;
- There seems to be no built-in support for guaranteeing protocol completion.

This report is structured as follows:

- Section 2 presents the **motivating example**;
- Section 3 presents our **detailed assessment**.

2 Motivating example

To make an assessment on Plural with respect to its ability to statically track the state of different objects inside a collection, we present the implementation of a linked-list collection and a file reader with a usage protocol. All code is available online[2]. We believe this example is relevant because linked-lists are common data structures. Furthermore, their use of pointers often creates challenges for less expressive type systems[3], so they are great candidates for use case examples.

The linked-list is single-linked, meaning that each node has a reference only to the next node. Internally, there are two fields, *head* and *tail*. The former points to the first node, the latter points to the last node. Items are added to the *tail* of the structure and removed from the *head*, following a FIFO discipline. The file reader has a usage protocol such that one must first call the *open* method, followed by any number of *read* calls until the end of the file is reached (which is checked by calling the *eof* method), and then terminated with the *close* method.

File reader implementation To model the file reader's protocol in Plural[4], we define four states, *init*, *opened*, and *closed*, which refine the root state *alive* (line 2 of List. 1). Additionally, we define two states, *eof* and *notEof*, which refine the *opened* state, indicating if we have reached or not the end of the file, respectively (line 3). The file reader has a boolean field called *remaining*, indicating if there is still something to read. If we have reached the end of the file, *remaining* is *false*, otherwise it is *true*. The invariants associated with the states *eof* and *notEof* ensure these properties (lines 8-9). Ideally, the *remaining* field would contain an integer value (with the number of bytes to read), unfortunately the syntax *remaining == 0* does not seem to be supported in the invariants. The *open*, *read*, and *close* methods require *unique* permission to the receiver object to change the state in accordance with the protocol. *Full* permissions would be enough except for the possibility of concurrent accesses, which would require the methods to be *synchronized*.[5] The *eof* method only needs *immutable* permission and returns a boolean value indicating if the end of the file was reached, assuming it was already opened (lines 34-42). The *if* statement (line 38) is required for Plural to understand that if *remaining* is *false*, we are in fact in the *eof* state. This implementation was accepted by Plural with no issues.

[2] `https://github.com/jdmota/tools-examples/tree/main/plural`

[3] For example, in Rust, one has to follow an ownership discipline, preventing one from creating linked-lists, unless *unsafe* code is used. GhostCell [8], a recent solution to deal with this, allows for internal sharing but the collection itself still needs to respect the ownership discipline. GhostCell uses *unsafe* code for its implementation but was proven safe with separation logic.

[4] `https://github.com/jdmota/tools-examples/blob/main/plural/FileReader.java`

[5] We could disable the synchronization checker, but we do not think that is the correct and safe approach.

Listing 1: File reader implementation

```
@Refine({
  @States(value={"init", "opened", "closed"}, refined="alive"),
  @States(value={"eof", "notEof"}, refined="opened")
})
@ClassStates({
  @State(name="init", inv="opened␣==␣false␣*␣closed␣==␣false"),
  @State(name="opened", inv="opened␣==␣true␣*␣closed␣==␣false"),
  @State(name="eof", inv="remaining␣==␣false"),
  @State(name="notEof", inv="remaining␣==␣true"),
  @State(name="closed", inv="opened␣==␣true␣*␣closed␣==␣true")
})
public class FileReader {
  private boolean remaining;
  private boolean opened, closed;

  @Perm(ensures="unique(this!fr)␣in␣init")
  public FileReader() {
    remaining = true;
    opened = false;
    closed = false;
  }

  @Unique(requires="init", ensures="opened", use=Use.FIELDS)
  public void open() {
    opened = true;
  }

  @Unique(requires="notEof", ensures="opened", use=Use.FIELDS)
  public byte read() {
    remaining = false;
    return 0;
  }

  @Imm(guarantee="opened", use=Use.FIELDS)
  @TrueIndicates("eof")
  @FalseIndicates("notEof")
  public boolean eof() {
    if (remaining == false) {
      return true;
    }
    return false;
  }

  @Unique(requires="eof", ensures="closed", use=Use.FIELDS)
  public void close() {
    closed = true;
  }
}
```

File reader examples In List. 2, we have an example of correct use of the file reader. In this scenario, Plural reports no errors, as expected.

Listing 2: Correct use of file reader

```
FileReader f = new FileReader();
f.open();
while (!f.eof()) f.read();
f.close();
```

In List. 2, we show an example of incorrect use of the file reader. Notice how the loop condition (line 3) is inverted and we are calling *read* when we reach the end of the file, instead of doing the opposite. Additionally, this causes *close* to be called before reaching the end of the file. Plural, as expected, reports these errors in lines 4 and 6.

Listing 3: Incorrect use of file reader

```
1 FileReader f = new FileReader();
2 f.open();
3 while (f.eof()) {
4   f.read(); // error: argument this must be in state [notEof] but is in [eof]
5 }
6 f.close(); // error: argument this must be in state [eof] but is in [notEof]
```

Finally, in List. 4, we have a method that expects to receive a file reader in the *opened* state and does not return it to the caller (notice the *returned* parameter in the annotation). In this scenario, Plural reports no errors even though we are "dropping" a permission to the file reader. As far as we can tell, Plural's specification language is based on linear logic, which would imply that this scenario would not be accepted. However, we understand why this is the case in Java, since it is common for Java programmers to stop using an object and letting the garbage collector reclaim memory. Nonetheless, we believe that ensuring protocol completion is crucial for typestated-objects, to ensure that necessary method calls are not forgotten and resources are freed (e.g. closing a socket). Unfortunately, protocol completion is not a guarantee that Plural seems to provide.

Listing 4: Dropping file reader

```
void droppingObject(@Unique(requires="opened", returned=false) FileReader f) {}
```

Linked-list implementation The linked-list requires a *Node*[6] and *LinkedList*[7] classes. These were adapted from a stack example provided in Plural's repository.[8] Naturally, we made the appropriate changes since our linked-list follows a FIFO discipline, while a stack follows a LIFO one.

To model each node we define two state dimensions, *dimValue* and *dimNext*, which handle the *value* and *next* fields, respectively (lines 3 and 4 of List. 5). In the *dimValue* dimension there are two states, *withValue* and *withoutValue*, which indicate if the node has permission to the stored value or not, respectively. In the *dimNext* dimension there are two states, *withNext* and *withoutNext*, which indicate if the node has permission to the next node or if *next* is *null*, respectively. The use of state dimensions was particularly useful to avoid the need to reason about all the combinations of having (or not) a value and having (or not) a next node. The invariants of each state are specified in lines 7 to 10.

The constructor accepts a (parametric) permission to a value and creates a node in states *withValue* and *withoutNext* (lines 17-21). The *getNext* method extracts the next node leaving the current one without a reference and permission to it (lines 23-30). The *setNext* method takes complete ownership of a node and sets it as the next node (lines 32-37). The *getValue* method gives all the permission it has to the value to the caller (lines 39-43). The *hasNext* method indicates if this node as a next node (lines 45-53).

[6] https://github.com/jdmota/tools-examples/blob/main/plural/Node.java

[7] https://github.com/jdmota/tools-examples/blob/main/plural/LinkedList.java

[8] File pluralism/trunk/PluralTestsAndExamples/src/edu/cmu/cs/plural/polymorphism/ecoop/Stack.java

Listing 5: Linked-list node implementation

```
@Similar("p")
@Refine({
  @States(value={"withValue", "withoutValue"}, dim="dimValue"),
  @States(value={"withNext", "withoutNext"}, dim="dimNext")
})
@ClassStates({
  @State(name="withValue", inv="p(value)"),
  @State(name="withoutValue", inv="true"),
  @State(name="withNext", inv="unique(next)␣in␣withValue,dimNext␣*␣next␣!=␣null"),
  @State(name="withoutNext", inv="next␣==␣null")
})
public class Node<T> {
  @Apply("p")
  private Node<T> next;
  private T value;

  @Perm(ensures="unique(this!fr)␣in␣withValue,withoutNext")
  public Node(@PolyVar(value="p", returned=false) T val) {
    value = val;
    next = null;
  }

  @Unique(requires="withNext", ensures="withoutNext", use=Use.FIELDS)
  @ResultUnique(ensures="withValue,dimNext")
  @ResultApply("p2")
  public Node<T> getNext() {
    Node<T> n = next;
    next = null;
    return n;
  }

  @Unique(requires="withoutNext", ensures="withNext", use=Use.FIELDS)
  public void setNext(
    @Unique(requires="withValue,dimNext", returned=false) @Apply("p") Node<T> n
  ) {
    next = n;
  }

  @Unique(requires="withValue", ensures="withoutValue", use=Use.FIELDS)
  @ResultPolyVar("p")
  public T getValue() {
    return value;
  }

  @Unique(guarantee="dimNext", use=Use.FIELDS)
  @TrueIndicates("withNext")
  @FalseIndicates("withoutNext")
  public boolean hasNext() {
    if (next == null) {
      return false;
    }
    return true;
  }
}
```

To model the linked-list we define two abstract states: the empty state and the non-empty state (line 3 of List. 6). When the list is empty, the *head* and *tail* field are *null* (line 6). When the list is not empty, *head* and *tail* are not *null* and there is unique permission to the first node, which is pointed by *head* (line 9). Since the *head* points to the next node, and so on, we should have the required chain of nodes that builds the linked-list. To add a new element to the list, we require some (parametric) permission to the value to be added, then we create a new node, and append it to the end of the list (lines 24-34). To remove an element from the list, we require the list to be non-empty, return the value stored in the *head* node, and make the second node the new *head* (lines 36-47). Both operations require *unique* permission to the list.

Listing 6: Linked-list implementation

```
 1  @Similar("p")
 2  @Refine({
 3    @States(value={"empty", "notEmpty"}, refined="alive")
 4  })
 5  @ClassStates({
 6    @State(name="empty", inv="head␣==␣null␣*␣tail␣==␣null"),
 7    @State(
 8      name="notEmpty",
 9      inv="unique(head)␣in␣withValue,dimNext␣*␣head␣!=␣null␣*␣tail␣!=␣null"
10    )
11  })
12  public class LinkedList<T> {
13    @Apply("p")
14    private Node<T> head;
15    @Apply("p")
16    private Node<T> tail;
17
18    @Perm(ensures="unique(this!fr)␣in␣empty")
19    public LinkedList() {
20      head = null;
21      tail = null;
22    }
23
24    @Unique(requires="alive", ensures="notEmpty", use=Use.FIELDS)
25    public void add(@PolyVar(value="p", returned=false) T value) {
26      @Apply("p") Node<T> n = new Node<T>(value);
27      if (head == null) {
28        head = n;
29        tail = n;
30      } else {
31        tail.setNext(n);
32        tail = n;
33      }
34    }
35
36    @Unique(requires="notEmpty", ensures="alive", use=Use.FIELDS)
37    @ResultPolyVar("p")
38    public T remove() {
39      T result = head.getValue();
40      if (head.hasNext()){
41        head = head.getNext();
42      } else {
43        head = null;
44        tail = null;
45      }
46      return result;
47    }
48  }
```

Unfortunately, Plural did not accept both implementations. With regards to the class *Node*, we had errors in all the methods indicating that the receiver could not be packed to match the state specified by the *ensures* annotation parameter. Additionally, the invariant for the *withValue* state (i.e. *p(value)*, line 7 of List. 5), had an error stating that the permission kind *p* was unknown, even though we introduced that parametric permission kind using the *@Similar* annotation (line 1). In fact, we did the same for the *LinkedList* class and we did not get any errors related to permissions kinds.

With regards to the *LinkedList* class, the only errors reported were in the *add* method. The *remove* method was fully accepted because we use *head.hasNext()* instead of *head != tail* to check if the list has more than one node, which more directly informs Plural that there is another node next to the *head*.

To understand why *add* was not accepted, consider the case in which the list is not empty. In this scenario, the *tail* is non-null, and we must call *setNext* on it to append a new node to the list (line 31 of List. 6). However, we need permission to do that and we do not have it. For this example to work, we would need to obtain the permission to the last node that is owned by the second to last node. We have an implementation of a linked-list in VeriFast[9], a modular verifier for C and Java programs annotated with specifications written in separation logic, but it required the definition of multiple predicates and lemmas. Since that does not seem to be supported by Plural, we think we cannot model the linked-list in this way.

An alternative solution could be to use *share* permissions instead of *unique* permissions in the nodes. Unfortunately, this would require locking when accessing them, because of the possibility of thread concurrency. Furthermore, we would lose track of the memory footprint used by the linked-list (since *share* permissions allow for unrestricted aliasing). This can be an issue if we want to track all the references and ensure statically that all resources are freed at end.

Even if implementing such a structure was possible, we would like to have a collection that was parametric on the typestates of the objects stored inside, so that we could track the state changes of these. But, as far as we can tell, even though parametric fractional permissions are supported, the typestates need to be uniform.

3 Assessment

Plural has the richest set of access permissions we know of, allowing the state of objects to be tracked even in the presence of aliasing, and permitting read/write and write/write operations, thanks to state guarantees. Nonetheless, we believe it is difficult to specify pointer-based data structures, such as linked-lists, where the last node is referenced from both the *next* field of the second to last node and the *tail* field. We believe the support for logical predicates would be useful for specifying structures with recursive properties, and avoiding the repetition of statements. Furthermore, as far as we can tell, there is no support for parametric typestates, even though there is for fractional permissions, which would allow one to model a list with objects in different states that evolve.

The use of *share* permissions allows for unrestricted aliasing. Nonetheless, state assumptions need to be discarded because of the possibility that there might be other threads attempting to modify the same reference. Although this thread-sharedness approximation is sound, it forces the use of synchronization primitives even if a reference is only available in one thread. Beckman et al. [2] discuss the possibility of distinguishing permissions for references that are only aliased locally from references that are shared between multiple threads, allowing access to thread-local ones without the need for synchronization. But as far as we know, the idea was not realized.

[9] `https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedList.java`

Furthermore, there seems to be no built-in support for guaranteeing protocol completion. This feature is important in typestate-oriented contexts because we want to ensure that necessary method calls are not forgotten and resources are freed. Nonetheless, such could be supported by asking the programmer to indicate which state should be the final state of a given object and allowing permissions (only) for *ended* objects to be "dropped".

With respect to the programming effort, we do not think it was very demanding. We spent some time trying to understand which were the correct annotations to use (from examples in Plural's source code), and thinking about how to model the protocol of each class, but besides that the specification effort was minimal. However, this may be due to the fact that the file reader was very straightforward to implement, and because we knew beforehand that we would not be able to implement the linked-list.

References

[1] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3 (2-3):95–230, 2016. doi: 10.1561/2500000031.

[2] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 227–244. ACM, 2008. doi: 10.1145/1449764.1449783.

[3] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 301–320. ACM, 2007. doi: 10.1145/1297027.1297050.

[4] José Duarte and António Ravara. Retrofitting Typestates into Rust. In *SBLP'21: 25th Brazilian Symposium on Programming Languages, 2021*, pages 83–91. ACM, 2021. doi: 10.1145/3475061.3475082.

[5] Oscar Nierstrasz. Regular types for active objects. *ACM sigplan Notices*, 28(10):1–15, 1993.

[6] André Trindade, João Mota, and António Ravara. Typestate Editor. `https://typestate-editor.github.io/`.

[7] André Trindade, João Mota, and António Ravara. Typestates to Automata and back: a tool. In *Proceedings 13th Interaction and Concurrency Experience, ICE 2020*, volume 324 of *EPTCS*, pages 25–42, 2020. doi: 10.4204/EPTCS.324.4.

[8] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi: 10.1145/3473597.