On using VeriFast to check object usage

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

May 11, 2022

Contents

1 Introduction

The goal of this report is twofold: (1) assess if VeriFast [6, 7] 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 [10, 1]. One might represent their intended usage protocol with an automaton or a state machine [12, 11, 3]. **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.

We build examples in Java and check them with the tool. Even though VeriFast supports both C and Java, we choose to work with Java because it is object-oriented and so, it is more suited for building objects with protocols where method calls are transitions.

We conclude that VeriFast is capable of verifying complex programs thanks to its specifications based on separation logic. Nonetheless, we find some drawbacks:

- Deductive reasoning via lemmas is often required, as well as the explicit *opening* and *closing* of predicates. This is tedious and can be a barrier to less experienced users;
- Fractional permissions only allow for read-only access when data is shared. In consequence, either locks are required to mutate shared data (even in single-threaded code, where they are not really necessary, resulting in inefficient code), or a complex specification workaround is needed;
- There is no built-in support for guaranteeing protocol completion.

This report is structured as follows:

- Section 2 presents the **motivating example**;
- Section 3 shows a **use example** of the classes explained in the previous section;
- Section 4 discusses an attempt to guarantee **protocol completion**;
- Section 5 discusses some **limitations of fractional permissions**;
- Section 6 presents our **detailed assessment**.

2 Motivating example

To make an assessment on VeriFast 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, an iterator for such collection, and a file reader with a usage protocol. Following that, we show an example where file readers are stored in a linked-list and then used according to their protocol. All code is available online[1]. 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[2], so they are great candidates for use case examples.

The linked-list[3] 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[4] 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, we use pre- and post-conditions in all public methods indicating the expected state and the destination state after the call. To require access to the object's fields and keep track of the current state, we define the

[1] `https://github.com/jdmota/tools-examples/tree/main/verifast`

[2] For example, in Rust, one has to follow an ownership discipline, preventing one from creating linked-lists, unless *unsafe* code is used. GhostCell [13], 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.

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

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

filereader predicate (List. 1). The field *remaining* is the number of bytes left to read. When it is 0, it means we reached the end of the file. The implementation is very straightforward[5].

Listing 1: *filereader* predicate

```
predicate filereader(FileReader file; int state, int remaining) =
  file.state |-> state &*& file.remaining |-> remaining &*& remaining >= 0;
```

Linked-list implementation The linked-list implementation[6] is adapted and extended from a C implementation available online[7]. One key difference from the aforementioned C code is that when the linked-list is empty, the *head* and *tail* fields have *null* values, instead of pointing to a dummy node.

To model the linked-list we use the *llist* predicate (List. 2). This predicate holds the heap chunks of the *head* and *tail* fields and of all the nodes in the list. The last parameter allows us to reason about the elements of the list in an abstract way. Lines 3 and 4 ensure that if one of the fields is *null*, the other is also *null* and the list is empty. To ease the addition of new elements to the list, which requires easy access to the tail node, we request access to the sequence of nodes between *head* (inclusive) and *tail* (exclusive), through the *lseg* predicate (List. 3), and then keep access to the *tail* directly in the body of the *llist* predicate (line 5).

Listing 2: *llist* predicate

```
1 predicate llist(LinkedList javalist; Node h, Node t, list<FileReader> list) =
2   javalist.head |-> h &*& javalist.tail |-> t &*&
3     h == null ? t == null &*& list == nil :
4     t == null ? h == null &*& list == nil :
5     lseg(h, t, ?l) &*& node(t, null, ?value) &*&
6       list == append(l, cons(value, nil)) &*& list != nil;
```

The *lseg* predicate (List. 3) holds the heap chunks of the nodes in the sequence starting on *n1* (inclusive) and ending on *n2* (exclusive), mapping to the elements on last parameter *list*.

Listing 3: *lseg* predicate

```
predicate lseg(Node n1, Node n2; list<FileReader> list) =
  n1 == n2 ?
    list == nil :
    node(n1, ?next, ?value) &*& lseg(next, n2, ?l) &*& list == cons(value, l);
```

The implementation of the *remove*[8] and *notEmpty*[9] methods is straightforward requiring only the opening and closing of the *llist* and *lseg* predicates a few times. The implementation of the *add*[10] method requires a lemma (List. 4) which states that if we have a sequence of nodes plus the final node, and we append another node in the end, we get a new sequence with all the nodes from the previous sequence, the previous final node, and then the newly appended node.

[5] https://github.com/jdmota/tools-examples/blob/main/verifast/basic/FileReader.java
[6] https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedList.java
[7] https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/iter.c.html
[8] https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedList.java#L60-L78
[9] https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedList.java#L80-L87
[10] https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedList.java#L42-L58

Listing 4: *add_lemma* lemma

```
lemma void add_lemma(Node n1, Node n2, Node n3)
  requires lseg(n1, n2, ?l) &*& node(n2, n3, ?value) &*& node(n3, ?n4, ?v);
  ensures lseg(n1, n3, append(l, cons(value, nil))) &*& node(n3, n4, v);
```

Iterator implementation To model the iterator[11] we use the *iterator* predicate (List. 5) which holds access to the current node in the *curr* field and all the nodes in the linked-list. This predicate has two final parameters which are the list of elements already read and the list of elements still to read, respectively. With the *iterator_base* predicate (List. 6), the permissions to the nodes are split in two parts. Half of the permissions "preserves the structure" of the list (line 4), and the other half holds the view of the iterator (line 5): a sequence of nodes from the *head* (inclusive) to the current node (exclusive), using the *lseg* predicate (List. 3); and a sequence from the current node (inclusive) to the final one, using the *nodes* predicate (List. 7).

Listing 5: *iterator* predicate

```
predicate iterator(LinkedList javalist, LinkedListIterator it, Node n;
  list<FileReader> list, list<FileReader> a, list<FileReader> b) =
  it.curr |-> n &*& iterator_base(javalist, n, list, a, b);
```

Listing 6: *iterator_base* predicate

```
1 predicate iterator_base(LinkedList javalist, Node n;
2   list<FileReader> list, list<FileReader> a, list<FileReader> b) =
3   [1/2]javalist.head |-> ?h &*& [1/2]javalist.tail |-> ?t &*&
4   [1/2]llist(javalist, h, t, list) &*&
5   [1/2]lseg(h, n, a) &*& [1/2]nodes(n, b) &*& list == append(a, b);
```

Listing 7: *nodes* predicate

```
predicate nodes(Node n; list<FileReader> list) =
  n == null ?
    list == nil :
    node(n, ?next, ?value) &*& nodes(next, ?l) &*& list == cons(value, l);
```

To create the iterator[12], we take the permission to all the nodes in the linked-list and split it in the aforementioned way. After iterating through all the nodes, the full permission to the nodes needs to be restored to the list. These actions are done with the *prepare_iterator* (List. 8) and the *dispose_iterator* (List. 9) lemmas, respectively. These requires some auxiliary lemmas, namely one showing that the result of appending a list with an element is not an empty list[13].

Listing 8: *prepare_iterator* lemma

```
lemma void prepare_iterator(LinkedList javalist)
  requires llist(javalist, ?h, ?t, ?list);
  ensures iterator_base(javalist, h, list, nil, list);
```

[11] `https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedListIterator.java`
[12] `https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedListIterator.java#L19-L26`
[13] `https://github.com/jdmota/tools-examples/blob/main/verifast/basic/Lemmas.java#L27-L35`

Listing 9: *dispose_iterator* lemma

```
lemma void dispose_iterator(LinkedListIterator it)
  requires iterator(?javalist, it, null, ?list, list, nil);
  ensures llist(javalist, _, _, list);
```

The implementation of the *hasNext* method[14] is straightforward and requires only the opening and closing of some predicates. The implementation of the *next* method[15] requires opening and closing predicates, the use of a lemma which shows that the *append* function is associative (result already available in VeriFast), and the *iterator_advance* lemma, which helps us advance the state of the iterator (List. 10).

Listing 10: *add_lemma_iterator* lemma

```
lemma void iterator_advance(Node h, Node n, Node t)
  requires [1/2]lseg(h, n, ?a) &*& [1/2]node(n, ?next, ?val1) &*&
    [1/2]nodes(next, ?b) &*& [1/2]lseg(h, t, ?list) &*& [1/2]node(t, null, ?val2);
  ensures [1/2]lseg(h, next, append(a, cons(val1, nil))) &*&
    [1/2]nodes(next, b) &*& [1/2]lseg(h, t, list) &*& [1/2]node(t, null, val2);
```

3 Use example

To exemplify the use of a linked-list with file readers, we instantiate a linked-list and add three file readers in the *Init* state to it (lines 1-7 of List. 11). Then we pass the list to the *useFiles* method (line 12) which iterates through all the files and executes their protocol to the end. To assert a fact about elements of the list we use the *foreachp* predicate provided by VeriFast, which automatically *closes* the invariant predicates for each file reader. However, we still need to explicitly *close* the *foreachp* predicate a few times (lines 9-11).

Listing 11: *Main* code

```
 1  LinkedList list = new LinkedList();
 2  FileReader f1 = new FileReader("a");
 3  FileReader f2 = new FileReader("b");
 4  FileReader f3 = new FileReader("c");
 5  list.add(f1);
 6  list.add(f2);
 7  list.add(f3);
 8  //@ assert llist(list, _, _, ?l);
 9  //@ close foreachp(cons(f3, nil), INV(FileReader.STATE_INIT));
10  //@ close foreachp(cons(f2, cons(f3, nil)), INV(FileReader.STATE_INIT));
11  //@ close foreachp(l, INV(FileReader.STATE_INIT));
12  useFiles(list);
```

The pre- and post-conditions of the *useFiles* method ensure that we get a list with all file readers in the *Init* state and we end up with the same list but with the readers in the *Closed* state (List. 12). This is possible because the predicate which models the iterator keeps track of the elements already seen and the elements to see.

[14] `https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedListIterator.java#L28-L38`

[15] `https://github.com/jdmota/tools-examples/blob/main/verifast/basic/LinkedListIterator.java#L40-L59`

Listing 12: *useFiles* contract

```
public static void useFiles(LinkedList list)
  //@ requires llist(list, _, _, ?l) &*& foreachp(l, INV(FileReader.STATE_INIT));
  //@ ensures llist(list, _, _, l) &*& foreachp(l, INV(FileReader.STATE_CLOSED));
```

The verification of the *useFiles* method (List. 13) requires loop invariants to be provided (lines 3-5 and line 13), the opening and closing of the *foreachp* predicate a few times (lines 19-20), the *foreachp_append* lemma provided by VeriFast (which says that if something is true of all elements of two lists, it is also true of the concatenation of the two) (line 21), and the *dispose_iterator* lemma (List. 9) (which restores the access of the nodes to the linked-list) (line 23).

Listing 13: *useFiles* code

```
 1  LinkedListIterator it = list.iterator();
 2  while (it.hasNext())
 3    /*@ invariant it != null &*& iterator(list, it, _, l, ?a, ?b) &*&
 4        foreachp(a, INV(FileReader.STATE_CLOSED)) &*&
 5        foreachp(b, INV(FileReader.STATE_INIT));
 6      @*/
 7  {
 8    FileReader f = it.next();
 9    //@ open foreachp(b, _);
10    //@ open INV(FileReader.STATE_INIT)(f);
11    f.open();
12    while (!f.eof())
13      //@ invariant f != null &*& filereader(f, FileReader.STATE_OPENED, _);
14    {
15      //@ open filereader(f, _, _);
16      f.read();
17    }
18    f.close();
19    //@ close foreachp(nil, INV(FileReader.STATE_CLOSED));
20    //@ close foreachp(cons(f, nil), INV(FileReader.STATE_CLOSED));
21    //@ foreachp_append(a, cons(f, nil));
22  }
23  //@ dispose_iterator(it);
```

4 Protocol completion

In an attempt to ensure that all file readers created through the lifetime of the program reach the end of their protocol, we define the *tracker* predicate in the *tracker.javaspec* file[16] which keeps hold of the number of open file readers. This proposed solution is based on a private exchange with Jacobs [5]. Then, we augment the file reader's specification[17] to increment this counter in the constructor (List. 14), and decrement the counter in the *close* method (List. 15). To effectively increment or decrement the counter, we use the *increment_tracker* and *decrement_tracker* lemmas in the implementation of the constructor and *close* method. Finally, we assert in the pre-condition and post-condition of the *main* that the counter should be 0 (List. 16)[18].

[16]https://github.com/jdmota/tools-examples/blob/main/verifast/rt/tracker.javaspec

[17]https://github.com/jdmota/tools-examples/blob/main/verifast/protocol-completion-2/FileReader.java

[18]https://github.com/jdmota/tools-examples/blob/main/verifast/protocol-completion-2/Main.java#L8-L9

Listing 14: FileReader's constructor

```
public FileReader(String filename)
  //@ requires tracker(?n);
  //@ ensures tracker(n + 1) &*& filereader(this, STATE_INIT, _);
{
  this.state = STATE_INIT;
  this.remaining = 20;
  //@ increment_tracker();
}
```

Listing 15: FileReader's *close* method

```
public void close()
  //@ requires tracker(?n) &*& filereader(this, STATE_OPENED, 0);
  //@ ensures tracker(n - 1) &*& filereader(this, STATE_CLOSED, 0);
{
  this.state = STATE_CLOSED;
  //@ decrement_tracker();
}
```

Listing 16: *main* method

```
public static void main(String[] args)
  //@ requires tracker(0);
  //@ ensures tracker(0);
```

Unfortunately, it is possible to fail to ensure protocol completion if the programmer is not careful. Firstly, one could forget to increment and decrement the counter when the typestated-object is initialized and when its protocol finishes, respectively. Secondly, if one forgets to add the post-condition to the *main* method, protocol completion will not be actually enforced. So, we can guarantee protocol completion but only if the programmer does not fall for these "traps". Here we see that ghost code is useful to check some properties, but if such code is not correctly connected with the "real" code, then the property we desired to establish is not actually guaranteed.

An alternative solution is to use a ghost list that tracks each opened file reader, instead of keeping a counter[19]. As with this solution, when a file is initialized, it needs to be added to the list, and when a file is closed, it needs to be removed from the list. Additionally, the *main* method requires a post-condition indicating that the ghost list is empty.

5 Sharing of mutable data

Consider a scenario where some callbacks are executed asynchronously in a single-threaded context. This kind of use case is very common in web applications and *Node.js*[20], and works thanks to an *event loop*, which is in charge of queuing and firing events without relying on multithreading (i.e. the event loop executes each callback at a time). Libraries such as *ActiveJ*[21] may be used to implement asynchronous operations in Java.

Suppose that we have two callbacks that are responsible for adding a new item to a linked-list. Clearly both callbacks needs exclusive access to the linked-list to modify it. Giving each callback access to the list at the point of initialization does not work because as soon as we split the

[19]https://github.com/jdmota/tools-examples/tree/main/verifast/protocol-completion-1
[20]https://nodejs.org/en/
[21]https://activej.io/

permission, we only get read-only access. One could use locks to ensure mutual exclusion when modifying the list[22], but that would create unnecessary overhead since the lack of parallelism already ensures exclusive accesses.

The solution we found[23], suggested by Jacobs [5], is to have a pool of objects that the event loop holds and provides to each callback at a time. For each callback to be able to find the linked-list in the pool, each holds 1/2 of the proof that the list is there. For simplicity, we implement an event loop that only allows two callbacks to be queued. For brevity, the listings containing Java code only show the methods' signature and contract.

For this implementation, we specify the *eventloop* predicate (lines 3-7 of List. 17) which holds access to the callbacks. Additionally, it keeps track of a ghost list containing predicate constructors and asserts that all the predicates hold, thanks to the combination of the *foreach* (from VeriFast's library) and the *holds* (line 1) predicates. We also define the *listCtor* predicate constructor which, when applied, asserts that we hold full access to a given linked-list (line 9). The *callback* predicate (lines 11-12) provides each callback access to its *list* field and a fraction of the proof that the list's memory footprint is available in the ghost list.

Listing 17: Event loop's predicates

```
1  predicate holds(predicate() p) = p();
2
3  predicate eventloop(EventLoop e, int id, list<predicate()> preds) =
4    e.cb1 |-> ?cb1 &*& e.cb2 |-> ?cb2 &*&
5    (cb1 == null ? emp : callback(cb1, id, _)) &*&
6    (cb2 == null ? emp : callback(cb2, id, _)) &*&
7    strong_ghost_list<predicate()>(id, preds) &*& foreach<predicate()>(preds, holds);
8
9  predicate_ctor listCtor(LinkedList l)() = l != null &*& llist(l, _, _, _);
10
11 predicate callback(Callback cb, int id, LinkedList l) =
12   cb.list |-> l &*& [_]strong_ghost_list_member_handle(id, listCtor(l));
```

The *EventLoop* class (List. 18) provides 3 essential methods: *addObject*, which inserts a new resource into the ghost list and provides a proof that such resource is in that list; *addCallback*, which schedules a new callback to be called; and *runAll*, which executes all queued callbacks, one at a time.

[22]https://github.com/jdmota/tools-examples/blob/main/verifast/callbacks/CallbackWithLocks.java
[23]https://github.com/jdmota/tools-examples/blob/main/verifast/callbacks/EventLoop.java

Listing 18: Event loop's implementation

```
 1 class EventLoop {
 2   public EventLoop()
 3     //@ requires true;
 4     //@ ensures eventloop(this, _, nil);
 5   { ... }
 6
 7   public void addObject(LinkedList l)
 8     //@ requires eventloop(this, ?id, ?preds) &*& listCtor(l)();
 9     /*@ ensures eventloop(this, id, append(preds, cons(listCtor(l), nil))) &*&
10                 strong_ghost_list_member_handle(id, listCtor(l)); */
11   { ... }
12
13   public void addCallback(Callback cb)
14     //@ requires eventloop(this, ?id, ?preds) &*& callback(cb, id, _);
15     //@ ensures eventloop(this, id, preds);
16   { ... }
17
18   public void runAll()
19     //@ requires eventloop(this, ?id, ?preds);
20     //@ ensures eventloop(this, id, preds);
21   { ... }
22 }
```

When a callback is initialized, it requires a fraction of the proof that the linked-list's memory footprint is present in the event loop's ghost list (line 3 of List. 19). This proof is kept in the *callback*, as we have seen. When the callback is executed, with the *run* method (line 7), the event loop provides its ghost list so that the callback can extract full access to the object it requires (line 8). By the end of the execution, the callback should return that access (line 9).

Listing 19: Callback's implementation

```
 1 class Callback {
 2   public Callback(LinkedList l)
 3     //@ requires [_]strong_ghost_list_member_handle(?id, listCtor(l));
 4     //@ ensures callback(this, id, l);
 5   { ... }
 6
 7   public void run()
 8     //@ requires callback(this, ?id, ?l) &*& eventloop(?e, id, ?preds);
 9     //@ ensures callback(this, id, l) &*& eventloop(e, id, preds);
10   { ... }
11 }
```

List. 20 presents a usage example of callbacks and the event loop. Initially, the shared linked-list and the event loop are initialized (lines 1-2). Then the memory footprint is provided to the event loop via the *addObject* method (line 4). Following that, two callbacks are initialized and queued in the event loop (lines 7-10). Finally, both callbacks are executed (line 12).

Listing 20: Callback's implementation

```
LinkedList l = new LinkedList();
EventLoop e = new EventLoop();
//@ close listCtor(l)();
e.addObject(l);

//@ split_fraction strong_ghost_list_member_handle(_, listCtor(l));
Callback3 c1 = new Callback3(l);
Callback3 c2 = new Callback3(l);
e.addCallback(c1);
e.addCallback(c2);

e.runAll();
```

Although we were able to successfully implement this use case, we believe this solution has some drawbacks. Firstly, we are forced to take all the resources the callbacks need (in this case, the linked-list) and put them explicitly in the pool of objects, which is cumbersome. Secondly, each callback needs to be aware of this pool and receive it in the pre-condition so that it can manipulate the required objects. In other words, instead of the event loop being abstracted away, its use is made explicit. We believe this highlights a very common scenario that occurs in tools that provide the possibility of deductive reasoning: even though it is usually possible to find a way to verify a complex application, the code needs to be implemented in such a way as to help the verifier check the specifications. Besides that, such specifications might require reasoning that is not related with the application's logic itself.

6 Assessment

VeriFast's use of, namely, separation logic, fractional permissions, and predicates, allows for rich and expressive specifications that make it possible to verify complex programs. However, deductive reasoning is often required when the specifications are more elaborate. For example, the opening and closing of predicates is necessary regularly, as well as the definition of multiple lemmas, as we have seen in the examples. This is tedious and can be a barrier to less experienced users. In our experience, we spent more time in proving results than in writing the code, having had to write about 160 lines of lemmas to make the linked-list and iterator implementations work.[24] However, implementing the file reader was very straightforward. Although VeriFast has an interactive IDE, which provides a way for one to observe each step of a proof, we believe that, at least in part, the use of a proof assistance (similar to Coq[25] for example) would improve the user experience even more. Nonetheless, this IDE was still very useful in helping us prove the aforementioned lemmas.

Although the support for both fractional and counting permissions is useful, these models only allow for read-only access when data is shared. In consequence, either locks are required to mutate shared data (even in single-threaded code, where they are not really necessary, resulting in inefficient code), or a complex specification workaround is needed. We believe that the specifications and code should focus on the application's logic, and the need to modify them to help the verifier should be avoided as much as possible.

In the context of typestates, checking for protocol completion is crucial to ensure that necessary method calls are not forgotten and that resources are freed, thus avoiding memory leaks. Un-

[24]https://github.com/jdmota/tools-examples/blob/main/verifast/basic/Lemmas.java
[25]https://coq.inria.fr/

fortunately, that concept is not built-in in separation logic. One solution we found is to have a counter that keeps track of all typestated-objects which are not in the final state. Whenever an object reaches the final state, the counter can be decremented. Then, we have a post-condition in the *main* method saying the counter should be 0. Of course, this requires keeping hold of the aforementioned tracker in specifications, which can be a huge burden in bigger programs. Moreover, if one forgets to add the post-condition to the *main* method, protocol completion will not be enforced. We believe that protocol completion should be provided directly by the type system and the programmer should not be required to remember to add this property to the specification. Ensuring protocol completion could be embedded in the logic and such feature could even be possible in VeriFast. Given that VeriFast supports leak checking, one would just need to incorporate notions of typestates, where one would specify what is the final state of an object, and ensure that leaking is only allowed when objects are in their final states. In C, one would also need to enforce that claimed memory is freed. In Java, leak checking would need to be enabled for typestated-objects.

Given the similarities with VerCors [4, 2], we believe it is also relevant to compare VeriFast with it. More details regarding VerCors may be found in a similar report about it[26], where we present the same experiments we show here.

With respect to specifying access to memory locations, VeriFast only supports the **points-to assertions** of separation logic, while VerCors also supports **permission annotations**, following the approach of Chalice [8, 9], allowing us to refer to values in variables without the need to use new names for them. Furthermore, VerCors has built-in support for quantifiers, many different abstract data structures, and ghost code, which VeriFast does not. Nonetheless, VeriFast supports the definition of new inductive data types, fixpoint functions, higher-order predicates, and counting permissions, which VerCors does not.

When considering the deductive reasoning often required, we noted that more interactive proofs would improve the user experience. VeriFast already provides an interactive experience, while VerCors does not, so the programmer has to practice "trial and error" constantly. In VerCors, just like in VeriFast, we spent more time in proving results about the linked-list (and iterator) than in writing the code, having had to write about 100 lines of lemmas.[27] Some of the time spent in VerCors with the proofs was reduced because we could reuse the experience we had with VeriFast. Unfortunately, some of the time gained was lost due to some issues, namely, the lack of support for output parameters from VeriFast, truth statements being lost when unfolding a permission in which whose statements depended, and somewhat confusing error messages.

References

[1] Davide Ancona et al. "Behavioral types in programming languages". In: *Foundations and Trends in Programming Languages* 3.2-3 (2016), pp. 95–230. DOI: `10.1561/2500000031`.

[2] Stefan Blom et al. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *Proceedings of Integrated Formal Methods, IFM 2017*. Vol. 10510. Lecture Notes in Computer Science. Springer, 2017, pp. 102–110. DOI: `10.1007/978-3-319-66845-1_7`.

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

[26]`https://github.com/jdmota/tools-examples/blob/main/vercors/on_using_vercors.pdf`

[27]`https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L31-L142`

[4] Marieke Huisman and Raúl E. Monti. “On the Industrial Application of Critical Software Verification with VerCors”. In: *Proceedings of Leveraging Applications of Formal Methods, ISoLA 2020.* Vol. 12478. Lecture Notes in Computer Science. Springer, 2020, pp. 273–292. DOI: 10.1007/978-3-030-61467-6_18.

[5] Bart Jacobs. Private communication. Mar. 2022.

[6] Bart Jacobs, Jan Smans, and Frank Piessens. “A Quick Tour of the VeriFast Program Verifier”. In: *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings.* Vol. 6461. Lecture Notes in Computer Science. Springer, 2010, pp. 304–311. DOI: 10.1007/978-3-642-17164-2_21.

[7] Bart Jacobs et al. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings.* Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

[8] K Rustan M Leino and Peter Müller. “A basis for verifying multi-threaded programs”. In: *European Symposium on Programming.* Springer. 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27.

[9] K Rustan M Leino, Peter Müller, and Jan Smans. “Verification of concurrent programs with Chalice”. In: *Foundations of Security Analysis and Design V.* Springer, 2009, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7.

[10] Oscar Nierstrasz. “Regular types for active objects”. In: *ACM sigplan Notices* 28.10 (1993), pp. 1–15.

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

[12] André Trindade, João Mota, and António Ravara. “Typestates to Automata and back: a tool”. In: *Proceedings 13th Interaction and Concurrency Experience, ICE 2020.* Vol. 324. EPTCS. 2020, pp. 25–42. DOI: 10.4204/EPTCS.324.4.

[13] Joshua Yanovski et al. “GhostCell: Separating Permissions from Data in Rust”. In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473597.