On using VerCors 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 VerCors [5, 2] 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 VerCors supports many different languages, namely, Java, C, OpenCL and PVL (Prototypal Verification Language), 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 VerCors 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 *folding* and *unfolding* 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;
- Counting permissions are not supported;
- Higher-order predicates are not supported;
- There is no built-in support for guaranteeing protocol completion.

This report is structured as follows:

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

2 Motivating example

To make an assessment on VerCors 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*[4]. 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[5] 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.

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

[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/vercors/LinkedList.java`

[4] In practice, the fields are named *h* and *t*, respectively, because *head* and *tail* are reserved words.

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

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 track the current state we use an integer ghost field (line 2 of List. 1). We tried to use constants, instead of explicitly using numbers, but there seems to be an internal error when static final fields are used in Java classes.[6] 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.[7]

Listing 1: *FileReader* class

```
1  public class FileReader {
2    //@ ghost private int state;
3    private int remaining;
4
5    //@ context Perm(state, 1) ** Perm(remaining, 1) ** remaining >= 0;
6    //@ requires state == 1;
7    //@ ensures state == 2 ** remaining == \old(remaining);
8    public boolean open() {
9      //@ ghost this.state = 2;
10     return true;
11   }
12
13   ...
14 }
```

Linked-list implementation To model the linked-list[8] we define in its class the *state* predicate (List. 2). This predicate holds the heap chunks of the *head* and *tail* fields and of all the nodes in the list. The single parameter allows us to reason about the elements of the list in an abstract way. Lines 4 to 6 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 *nodes_until* predicate (List. 3), and then keep access to the *tail* directly in the body of the *state* predicate (line 8).

Listing 2: *state* predicate of the linked-list

```
1  final resource state(seq<FileReader> list) =
2    Perm(h, 1) ** Perm(t, 1) **
3    (
4      h == null ? (t == null ** list == seq<FileReader> {}) :
5        (
6          t == null ? (h == null ** list == seq<FileReader> {}) :
7            (
8              nodes_until(h, t) ** PointsTo(t.next, 1, null) ** Perm(t.value, 1) **
9              list == (nodes_until_list(h, t) + seq<FileReader> { t.value })
10           )
11       )
12   );
```

The *nodes_until* predicate (List. 3) holds the heap chunks of the nodes in the sequence starting on *n* (inclusive) and ending on *end* (exclusive). The *nodes_until_list* method is used to reason

[6] "[abort] (class vct.col.rewrite.Flatten)At file FileReader.java from line 2 column 23 until line 2 column 41: internal error: current block is null"

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

[8] `https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java`

about the values that are stored in this sequence of nodes (List. 4). Notice how *nodes_until_list* requires access to the heap chunk of nodes in the sequence from *n* to *end* (line 1 of List. 4).

Listing 3: *nodes_until* predicate

```
static resource nodes_until(Node n, Node end) =
  n == end ?
    true :
    n != null ** Perm(n.next, 1) ** Perm(n.value, 1) ** nodes_until(n.next, end);
```

Listing 4: *nodes_until_list* method

```
1 requires [1\2]nodes_until(n, end);
2 static pure seq<FileReader> nodes_until_list(Node n, Node end) =
3   \unfolding [1\2]nodes_until(n, end) \in (
4     n == end ?
5       seq<FileReader> {} :
6       seq<FileReader> { n.value } + nodes_until_list(n.next, end)
7   );
```

The implementation of the *remove*[9] and *notEmpty*[10] methods is straightforward requiring only the unfolding and folding of the foreamentioned predicates a few times. The implementation of the *add*[11] method requires a lemma (List. 5) stating 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.

Listing 5: *add_lemma* lemma

```
requires n1 != null ** n2 != null ** n3 != null;
requires nodes_until(n1, n2) ** Perm(n2.next, 1) ** Perm(n2.value, 1);
requires n2.next == n3 ** PointsTo(n3.next, 1, null);
requires list == nodes_until_list(n1, n2) + seq<FileReader> { n2.value };
ensures nodes_until(n1, n3) ** PointsTo(n3.next, 1, null) ** list == nodes_until_list(n1, n3);
ghost static void add_lemma(Node n1, Node n2, Node n3, seq<FileReader> list) {
  ...
}
```

Iterator implementation To model the iterator[12] we define in its class the *state* predicate (List. 6) 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. In this predicate, 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: a sequence of nodes from the *head* (inclusive) to the current node (exclusive), using the *nodes_until* predicate (line 5); and a sequence from the current node (inclusive) to the final one, using the *nodes* predicate (line 6) (List. 7). For practical reasons, the *head* and *tail* are also referenced from the *first* and *last* ghost fields of the iterator (line 10).

[9] `https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L183-L207`
[10] `https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L209-L217`
[11] `https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L158-L181`
[12] `https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedListIterator.java`

Listing 6: *state* predicate of the iterator

```
final resource state(LinkedList linkedlist, seq<FileReader> s, seq<FileReader> r) =
  Perm(first, 1) ** Perm(last, 1) ** Perm(curr, 1) **
  Perm(linkedlist.h, 1\2) ** Perm(linkedlist.t, 1\2) **
  ([1\2]linkedlist.state(s + r)) **
  ([1\2]LinkedList.nodes_until(first, curr)) **
  ([1\2]LinkedList.nodes(curr)) **
  s == LinkedList.nodes_until_list(first, curr) **
  r == LinkedList.nodes_list(curr) **
  (\unfolding [1\2]linkedlist.state(s + r) \in
    (first == linkedlist.h ** last == linkedlist.t));
```

Listing 7: *nodes* predicate

```
static resource nodes(Node n) =
  n == null ? true : Perm(n.next, 1) ** Perm(n.value, 1) ** nodes(n.next);
```

To create the iterator[13], 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.

Listing 8: *prepare_iterator* lemma

```
requires ([1\2]nodes_until(h, t)) ** PointsTo(t.next, 1\2, null) ** Perm(t.value, 1\2);
requires list == nodes_until_list(h, t) + seq<FileReader> { t.value };
ensures [1\2]nodes(h);
ensures list == nodes_list(h);
ghost static void prepare_iterator(Node h, Node t, seq<FileReader> list) {
  ...
}
```

Listing 9: *dispose_iterator* lemma

```
requires ([1\2]nodes_until(h, t)) ** PointsTo(t.next, 1\2, null) ** Perm(t.value, 1\2);
requires list == nodes_until_list(h, t) + seq<FileReader> { t.value };
requires [1\2]nodes_until(h, null);
requires list == nodes_until_list(h, null);
ensures nodes_until(h, t) ** PointsTo(t.next, 1, null) ** Perm(t.value, 1);
ensures list == nodes_until_list(h, t) + seq<FileReader> { t.value };
ghost static void dispose_iterator(Node h, Node t, seq<FileReader> list) {
  ...
}
```

The implementation of the *hasNext* method[14] is straightforward and requires only the unfolding and folding of the *state* predicate. The implementation of the *next* method[15] requires unfolding and folding predicates, and the *advance* lemma, which helps us advance the state of the iterator, moving the just retrieved value from the remaining list to the seen list[16].

[13] https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L219-L232
[14] https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedListIterator.java#L52-L62
[15] https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedListIterator.java#L64-L82
[16] https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L103-L114

3 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 a *FileTracker*[17] which keeps hold of the number of open file readers using ghost code. Then, we augment the file reader's implementation to increment this counter in the constructor (List. 10), and decrement the counter in the *close* method (List. 11). Notice how the tracker is given to the constructor and *close* method as a ghost parameter through the *given* directive (line 1 of both listings). Finally, we assert in the pre-condition and post-condition of the *main* that the counter should be 0 (List. 12).

Listing 10: FileReader's constructor

```
1 //@ given FileTracker tracker;
2 //@ context Perm(tracker.active, 1);
3 //@ ensures Perm(state, 1) ** Perm(remaining, 1) ** state == 1 ** remaining >= 0;
4 //@ ensures tracker.active == \old(tracker.active) + 1;
5 public FileReader() {
6   //@ ghost this.state = 1;
7   this.remaining = 20;
8   //@ ghost tracker.inc();
9 }
```

Listing 11: FileReader's *close* method

```
1 //@ given FileTracker tracker;
2 //@ context Perm(tracker.active, 1);
3 //@ context Perm(state, 1) ** Perm(remaining, 1) ** remaining >= 0;
4 //@ requires state == 2 ** remaining == 0;
5 //@ ensures state == 3;
6 //@ ensures tracker.active == \old(tracker.active) - 1;
7 public void close() {
8   //@ ghost this.state = 3;
9   //@ ghost tracker.dec();
10 }
```

Listing 12: *main* method

```
//@ given FileTracker tracker;
//@ context Perm(tracker.active, 1);
//@ requires tracker.active == 0;
//@ ensures tracker.active == 0;
public static void main(String[] args) {
  ...
}
```

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.

[17] https://github.com/jdmota/tools-examples/blob/main/vercors/FileTracker.java

4 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 initial state to it (lines 1-8 of List. 13). Then we pass the list to the *useFiles* method (line 9) which iterates through all the files and executes their protocol to the end.

Listing 13: *Main* code

```
1 //@ ghost seq<FileReader> l;
2 LinkedList list = new LinkedList() /*@ then {l=newList;} @*/;
3 FileReader f1 = new FileReader() /*@ with {tracker=tracker;} @*/;
4 FileReader f2 = new FileReader() /*@ with {tracker=tracker;} @*/;
5 FileReader f3 = new FileReader() /*@ with {tracker=tracker;} @*/;
6 list.add(f1) /*@ with {oldList=l;} then {l=newList;} @*/;
7 list.add(f2) /*@ with {oldList=l;} then {l=newList;} @*/;
8 list.add(f3) /*@ with {oldList=l;} then {l=newList;} @*/;
9 useFiles(list) /*@ with {tracker=tracker; l=l;} @*/;
```

The pre- and post-conditions of the *useFiles* method ensure that we get a list with all file readers in the initial state (line 7 of List. 14) and we end up with the same list but with the readers in the final state (line 8). This is possible because the predicate which models the iterator keeps track of the elements already seen and the elements to see.

Listing 14: *useFiles* contract

```
1 //@ given FileTracker tracker;
2 //@ given seq<FileReader> l;
3 //@ context_everywhere Perm(tracker.active, 1);
4 //@ requires tracker.active == |l|;
5 //@ ensures tracker.active == 0;
6 //@ context linkedlist != null ** linkedlist.state(l);
7 //@ requires (\forall* int i; i >= 0 && i < |l|; PointsTo(l[i].state, 1, 1));
8 //@ ensures (\forall* int i; i >= 0 && i < |l|; PointsTo(l[i].state, 1, 3));
9 //@ context (\forall* int i; i >= 0 && i < |l|; Perm(l[i].remaining, 1) ** l[i].remaining >= 0);
10 public static void useFiles(LinkedList linkedlist)
```

Listing 15: *filereader* predicate

```
1 static resource filereader(FileReader f, int state) =
2   PointsTo(f.state, 1, state) ** Perm(f.remaining, 1) ** f.remaining >= 0;
```

One may notice that in the contract of the *useFiles* methods, we explicitly request access to the fields *state* and *remaining* of each of the file readers in the linked-list (lines 7-9), instead of hiding these implementation details with, for example, a *filereader* predicate (List. 15). Unfortunately, VerCors is not able to establish that all file readers in the list are in the initial state if we use such predicate in a *forall* directive. So, the workaround is to keep this predicate unfolded, resulting in more verbose specifications.

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*[18], and works thanks

[18] https://nodejs.org/en/

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

A clever solution would be to have a pool of objects that the event loop holds and provides to each callback at a time[20]. Then, when each callback is called, it would extract the list from the pool, use it, and then return it to the pool. To allow the pool to be generic over different types of objects, each callback would hold 1/2 of the proof that the list is in the pool. Unfortunately, as far as we can tell, there is no support for higher-order predicates, which we would need to store the permissions to the objects in the pool.

In any case, this solution would have some drawbacks. Firstly, we would be 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 would be cumbersome. Secondly, each callback would need to be aware of this pool and receive it in the pre-condition so that it could manipulate the required objects. In other words, instead of the event loop being abstracted away, its use would be 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

VerCors' use of, namely, separation logic, fractional permissions, and predicates, allows for rich and expressive specifications that make it possible to verify complex programs. However, higher-order predicates are not supported. Moreover, deductive reasoning is often required when the specifications are more elaborate. For example, the unfolding and folding 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 100 lines of lemmas to make the linked-list and iterator implementations work.[21] However, implementing the file reader was very straightforward. We believe that, at least in part, the use of a proof assistance (similar to Coq for example) would improve the user experience by allowing for interactive proofs.

Although the support for fractional permissions is useful, this model only allows 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

[19] `https://activej.io/`

[20] We did in fact implement this solution in VeriFast: `https://github.com/jdmota/tools-examples/blob/main/verifast/callbacks/EventLoop.java`

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

be avoided as much as possible. Besides that, the support for counting permissions is lacking, which would allow permissions to be split in different ways.

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. Unfortunately, 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.

Given the similarities with VeriFast [6, 7], we believe it is also relevant to compare VerCors with it. More details regarding VeriFast may be found in a similar report about it[22], 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 a way for one to observe each step of a proof, and we believe VerCors would benefit from something similar, to avoid the need to practice “trial and error” constantly. Additionally, allowing for more separation between lemma and predicate functions from code, instead of forcing these to belong to classes as static methods, would help improve readability, as others have also noted [4]. In VeriFast, just like in VerCors, we spent more time in proving results about the linked-list (and iterator) than in writing the code, having had to write about 160 lines of lemmas.[23] 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 issues we explain next.

We missed the support for output parameters from VeriFast. Instead, we had to add ghost parameters in many methods and explicitly pass values for those parameters when calling such methods. For example, when working with the linked-list, we kept track of the sequence of values in the list through ghost code, and always had to pass that sequence to each called method.[24] We also noticed that if we unfolded a predicate that is required to ensure some other truth statement, that truth statement would be lost. To workaround this, we had to use fractional permissions to keep hold of some fraction of the original predicate, and only unfold the other fractional part.[25] Additionally, the need for unfolding predicates inline seems cumbersome.

[22]`https://github.com/jdmota/tools-examples/blob/main/verifast/on_using_verifast.pdf`
[23]`https://github.com/jdmota/tools-examples/blob/main/verifast/basic/Lemmas.java`
[24]`https://github.com/jdmota/tools-examples/blob/main/vercors/Main.java#L12-L14`
[25]`https://github.com/jdmota/tools-examples/blob/main/vercors/LinkedList.java#L132`

In our opinion, we think the error messages could be more helpful[26], and the tool could be faster. Since the specifications need to be self-framing and the checking process is modular, VerCors could make use of caching to avoid re-checking parts of the code that were not modified.

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`.

[4] J.P. Hollander. *Verification of a model checking algorithm in VerCors*. Aug. 2021. URL: `http://essay.utwente.nl/88268/`.

[5] 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`.

[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`.

[26] `https://github.com/utwente-fmt/vercors/issues/479`