On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage

João Mota
NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal

Marco Giunti
NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal
School of Computing, Engineering & Digital Technologies, Teesside University, Middlesbrough, UK

António Ravara
NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal

Abstract

Typestates are a notion of behavioral types that describe protocols for stateful objects, specifying the available methods for each state. Ensuring methods are called in the correct order (protocol compliance), and that, if and when the program terminates, all objects are in the final state (protocol completion) is crucial to write better and safer programs. Objects of this kind are commonly shared among different clients or stored in collections, which may also be shared. However, statically checking protocol compliance and completion when objects are shared is challenging. To evaluate the support given by state of the art verification tools in checking the correct use of shared objects with protocol, we present a survey on four tools for Java: VeriFast, VerCors, Plural, and KeY. We describe the implementation of a file reader, linked-list, and iterator, check for each tool its ability to statically guarantee protocol compliance and completion, even when objects are shared in collections, and evaluate the programmer's effort in making the code acceptable to these tools. With this study, we motivate the need for lightweight methods to verify the presented kinds of programs.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Logic and verification; Theory of computation → Separation logic

Keywords and phrases Java, Typestates, VeriFast, VerCors, Plural, KeY

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.40

Category Experience Paper

Related Version *Extended Version*: `https://arxiv.org/abs/2209.05136`

Supplementary Material
Software: `https://github.com/jdmota/tools-examples/tree/ecoop-2023`
archived at `swh:1:dir:9b9f9f7a269c44c04c57d5f66ff27884de02d4bc`

Funding Partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI) and NOVA LINCS (UIDB/04516/2020).
João Mota: Partially supported by FCT.IP (2021.05297.BD).
Marco Giunti: Partially supported by Dstl, reference: ACC2028868.

Acknowledgements We would like to thank several members of the developer teams for the detailed responses and enlightening discussions, in particular, Bart Jacobs (VeriFast), Marieke Huisman (VerCors), Lukas Armborst (VerCors), Reiner Hähnle (KeY), Eduard Kamburjan (KeY), and Richard Bubel (KeY). Their feedback was indispensable for the development of this study.

1 Introduction

In object-oriented programming, one naturally defines objects where their methods' availability depends on their internal state [49]. For example, the `next` method of an iterator can only be called if there are items to be retrieved, otherwise it throws an exception. One might represent their intended usage protocol with an automaton or a state machine [60, 61, 25]. **Behavioral types** [34, 3], when used for object-oriented languages, allow us to statically check if all code of a program respects the protocol of each object, helping us to write safer programs with fewer errors. The crucial properties verified are **protocol compliance**, ensuring methods are called in the correct order, and **protocol completion**, guaranteeing that if and when the program terminates, all the objects are in their final states, ensuring required method calls are not forgotten, and that resources are freed, for example, that we close all sockets. Bravetti et al. present a formal treatment of these properties [19].

In session types approaches, objects 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 [32, 34]. Given that **sharing of objects** is very common, it should be naturally supported, without putting too much burden on the programmer. For example, pointer-based data structures, like linked-lists, usually rely on internal sharing (i.e. aliasing). Such collections may also be used to store an arbitrary number of objects in different states which need to be tracked. Developing these data-types, and applications using them, is often challenging. To our knowledge, there is no typestate-oriented support for tracking the states of objects in collections, while ensuring protocol compliance and completion. Given this, the present study has the objective of answering the following research question (in Section 3.4)

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

To study the contributions and limitations of the state of the art, we report our experience in verifying protocol compliance and completion with four tools for Java: VeriFast [39, 38], VerCors [33, 13], Plural [10], and KeY [1]. We picked these because of their rich features for verification, and because they are actively maintained (with the exception of Plural). We believe these cover the most used static analysis techniques which can be instructed to perform typestate verification. *VeriFast* checks programs annotated with method contracts written in **separation logic** [51, 55]. *VerCors*, however, uses **permission-based concurrent separation logic** to check programs, inspired by Chalice [44, 45]. *Plural* verifies that the protocols of objects are respected with **typestates** [59]. It introduces **access permissions** which combine typestate and object aliasing information, allowing state to be tracked and modified even when objects are shared, allowing for more uses beyond the "single writer vs multiple readers" model of fractional permissions [17]. *KeY* verifies sequential Java programs with specifications written in JML [43], based on first-order Java **dynamic logic** (JavaDL) [27]. It supports a great number of Java features and provides an interactive theorem prover with a high degree of automation and useful tactics to guide proofs. OpenJML is another verification tool based on JML [22]. Given that the differences [16] are not significant for our use case, and since KeY has an interactive prover, we focus our study on KeY.

To our knowledge, no such comparison study was previously done that focused on protocol compliance and completion. As we will observe, the running examples may look simple but constitute real challenges to these tools, especially since three of them are based on contracts, not behavioral types. Our conclusions support advocating for *lightweight verification methods* directed at these protocol related properties. The contributions of this paper are:

- Code **implementations** and **examples** of using file readers, linked-lists, and iterators (when possible), similar in all four tools, and **specifications** appropriate to each tool to verify the desired properties;
- An assessment if the tools can check **protocol compliance**, and if they can guarantee **protocol completion**, even with objects **shared in collections**;
- An evaluation of the **programmer's effort** in making the code examples acceptable to each tool, justifying the need for lightweight methods to verify these kinds of programs.

With regards to our level of experience, we are knowledgeable in the concepts used, having applied them in different settings, and have had practical experience with VeriFast before conducting this study. Regarding the other tools, we report our experience in using them for the first time. To validate this study, our assessments were shared with the development teams. From the responses we got, they found the examples to be interesting and challenging. All the received feedback was very valuable in helping us to refine and confirm our conclusions.

This paper is structured as follows: Section 2 provides an overview of each tool; Section 3 presents **code implementations and specifications**, and if they were **accepted**, thus reporting our experience and answering the **RQ** (Section 3.4); Section 4 discusses our **assessments**; Section 5 discusses relevant work; finally, Section 6 presents our **conclusions**.

2 Background

2.1 VeriFast

VeriFast is a modular verifier for (subsets of) C and Java programs annotated with method contracts (pre- and post-conditions) written in **separation logic** [39, 38, 51, 55]. Besides the points-to assertions from separation logic, specifications support the definition of inductive data types, predicates, and fixpoint functions. Additionally, **deductive reasoning** [5] is supported via the definition of lemmas and the insertion of assertions in key program points to guide verification. Furthermore, it comes with an IDE allowing one to observe each step of a proof when an error is encountered. VeriFast is then able to statically check that contracts are respected during execution and that programs will not raise errors such as null pointer exceptions or perform incorrect actions such as accessing illegal memory. Nonetheless, VeriFast's support for generics is still limited.[1]

When declaring predicates, one may also specify **output parameters**. These appear after input parameters separated with a semicolon. Output parameters need to be precisely defined in the predicate definition and allow its user to "extract" them. In Listing 1, the parameter `b` is defined as an output parameter of the `account` predicate (line 1). It is precisely defined to be the value of the `balance` field of the account object. The `|->` symbol represents the points-to assertion while `&*&` represents the separating conjunction binary operator. This balance can then be "extracted" using the `?` symbol in a pre-condition (line 5), existentially quantifying the name `b`, which can then be mentioned in the post-condition (line 6), indicating the effect of a deposit given the current balance `b`.

Listing 1 Output parameters example.

```
1 //@ predicate account(Account a; int b) = a.balance |-> b &*& b >= 0;
2 class Account {
3   int balance;
4   void deposit(int value)
```

[1] https://github.com/verifast/verifast/issues/271

```
5       //@ requires account(this, ?b) &*& 0 < value;
6       //@ ensures account(this, b + value);
7     {
8       balance += value;
9     }
10  }
```

For the sharing of memory locations, VeriFast has built-in support for **fractional permissions**, associating a number coefficient between 0 (exclusive) and 1 (inclusive) to each heap chunk [17]. By default, the coefficient is 1, which allows reads from and writes to a memory location. A number less than 1 allows only for reads. The programmer may provide coefficient patterns in the form of expressions, such as literal numbers or variables, or in the form of existentially quantified names (like `f` in line 2 of Listing 2). These patterns may be applied to points-to assertions but also to predicates. Applying a coefficient to a predicate is equivalent to multiplying it by each coefficient of each heap chunk mentioned in the predicate's body. Additionally, VeriFast supports the automatic splitting and merging of fractional permissions. Counting permissions are also supported via a trusted library [17].

Listing 2 Coefficient pattern example.

```
1  int getBalance()
2    //@ requires [?f]account(this, _);
3    //@ ensures [f]account(this, result);
4  { return balance; }
```

For C programs, VeriFast also supports leak checking: after consuming post-conditions, the heap must be empty. However, the programmer can leak certain resources with the `leak` command. For Java programs, leaking is always allowed.

2.2 VerCors

VerCors is a verifier for concurrent programs written in Java, C, OpenCL and PVL (Prototypal Verification Language), and annotated with method contracts [33, 13]. The specifications employ a logic based on **permission-based concurrent separation logic** [50]. The verification procedure is modular, checking each method in isolation given a contract with pre- and post-conditions. As in VeriFast, lemmas can be defined and assertions introduced to guide verification. Although there are other tools that perform static verification on annotated programs, VerCors focuses on supporting different concurrency patterns of high-level languages, and is designed to be language-independent. Support for inheritance and exceptions is still being worked upon based on theoretical work by Rubbens [56].[2] Internally, VerCors uses the Viper backend [48], which in turn uses Z3 [23].

VerCors supports two styles for specifying access to memory locations: **permission annotations**, following the approach of Chalice [44, 45]; and **points-to assertions** of separation logic, as in VeriFast. Both styles of specification have been shown to be equivalent by Parkinson and Summers [53]. The equivalence is presented in Figure 1. On the left-hand-side it is shown the use of a permission annotation, `Perm`, to request access to variable `var`, with fractional permission `p`, and storing a value equal to `val`. The `**` symbol represents the separating conjunction operator. On the right-hand-side it is shown the equivalent `PointsTo` assertion. Permission annotations are very useful because they allow us to refer to values in variables without the need to use new names for them.

[2] `https://vercors.ewi.utwente.nl/wiki/#inheritance-1`

```
Perm(var, p) ** var == val ≡ PointsTo(var, p, val)
```

Figure 1 Permission annotations equivalent to points-to assertions.

As an example, List. 3 shows a method contract which requires exclusive permission to write in field `val` (line 2), and ensures that the new value is equal to the old one incremented by one (line 3).

Listing 3 Permissions example.

```
1  /*@
2    requires Perm(val, 1);
3    ensures  Perm(val, 1) ** val == \old(val) + 1;
4  @*/
5  void increment(){
6    val = val + 1;
7  }
```

VerCors also has support for **ghost code**, including ghost parameters and results in methods. These are declared in methods' contracts with the `given` and `yields` keywords, respectively. When calling a method, one uses the `with` and `then` keywords to assign ghost parameters and retrieve return values, respectively. These features are exemplified in Listing 4 where a `sum` method yields a ghost result given two ghost parameters, `x` and `y`. Line 13 shows how this method may be used. Ghost code is useful to keep track of intermediate results, which only exist for the purpose of verification.

Listing 4 Ghost code example.

```
1  /*@
2    given int x;
3    given int y;
4    yields int res;
5    ensures res == x + y;
6  @*/
7  void sum() {
8    //@ ghost res = x + y;
9  }
10
11 void main() {
12   //@ ghost int result;
13   sum() /*@ with {x=3; y=2;} then {result=res;} @*/;
14 }
```

2.3 Plural

Bierhoff and Aldrich addressed the problem of substitutability of subtypes while guaranteeing **behavioral subtyping** in a object-oriented language [9]. The specification technique models protocols using abstract states, incorporating *state refinements* (allowing the definition of substates, thus supporting substitutability of subtypes), *state dimensions* (which define orthogonal states corresponding to AND-states in Statecharts [28]), and *method refinements* (allowing methods in subclasses to accept more inputs and return more specific results). The approach is similar to pre- and post-condition based ones but provides better information hiding thanks to the **typestate** abstraction [59].

Bierhoff and Aldrich then built on previous work and developed a sound modular protocol checking approach, based on typestates, to ensure at compile-time that clients follow the usage protocols of objects even in the presence of aliasing [10]. For that, they developed the

notion of **access permissions** which combine typestate and object aliasing information. The approach was realized in Plural, a static verifier they developed for Java, as a plugin for Eclipse. As far as we know, not all Java's features are supported, such as exceptions. Although it is not maintained any longer, one can install it in Eclipse Juno (an old version from 2012) from its source. Given its support for rich access permissions, and direct application of the typestate abstraction, we believe its study is still very relevant.

An *access permission* tracks how a reference is allowed to read and/or modify the referenced object, how the object might be accessed through other references, and what is currently known about the object's typestate. To increase the precision of access permissions, Bierhoff and Aldrich introduced *weak permissions* (such as *share* and *pure*), where an object can be modified through other permissions. The proposed permissions include a *state guarantee* which ensures that an object remains in that state even in the face of interference. Additionally, they track *temporary state assumptions* which are discarded when they become outdated. All kinds of permissions are described in Table 1. A comprehensive survey on permission-based specifications was presented by Sadiq et al. [57].

Table 1 Permissions in Plural.

Kind	Access to the referenced object	Access other aliases may have
Full	read and write	read-only
Pure	read-only	read and write
Immutable	read-only	read-only
Unique	read and write	none
Shared	read and write	read and write

In Listing 5 is an example of an iterator. In lines 1-3, it is stated that this object may be in two distinct states, `available` or `end`. These are defined as refinements, or subtypes, of the root state `alive`, a state common to all objects. Then it is specified that the `hasNext` method may be called in the `alive` state with just `pure` permission (line 5). If the method returns `true`, we further know that the iterator is in the `available` state, allowing us to refine our knowledge (line 6). Otherwise, the iterator is in the `end` state (line 7). The `next` method requires `full` permission and that the object be in the `available` state, and then it can only ensure it is in the `alive` state (line 10).

Listing 5 Iterator example.

```
1  @Refine({
2    @States(value={"available", "end"}, refined="alive")
3  })
4  interface Iterator<E> {
5    @Pure("alive")
6    @TrueIndicates("available")
7    @FalseIndicates("end")
8    boolean hasNext();
9
10   @Full(requires="available", ensures="alive")
11   E next();
12 }
```

These kinds of permissions may be split to allow sharing of an object, and joined back together to allow one to potentially restore *unique* permission. **Fractional permissions** are used to track how much a permission was split [18]. Furthermore, different fractions can be mapped to different state guarantees through a *fraction function*, thus tracking for each state guarantee separately how many other permissions rely on it.

Beckman et al. extended the approach to verify the correctness of usage protocols in concurrent programs, statically preventing races on the abstract state of an object as well as preventing violations of state invariants [7]. This approach uses atomic blocks and was also realized in Plural. In this solution, access permissions are used as an approximation of the thread-sharedness of objects. For example, if *pure* or *share* permissions are used, it means that other references can modify the object, and it is assumed that this includes concurrent modifications. In this scenario, temporary state assumptions are discarded, unless the access is synchronized. Furthermore, accessing fields of an object with *share*, *pure*, or *full* permissions, must be performed inside atomic blocks.

Beckman later presented a similar approach which uses synchronization blocks instead of atomic blocks as the mutual exclusion primitive, given that the former are in more wide use [6]. Since programmers are required to synchronize on the receiver object, it becomes implicit to the analysis which parts of the memory are exclusively available, so programmers are not required to specify which parts of the memory are protected by which locks. However, this also implies that private objects cannot be used for the purposes of mutual exclusion.

2.4 KeY

KeY is a verifier for sequential Java programs [1]. Specifications are provided in Java comments in JML*, an extension of the Java Modeling Language (JML) [43]. JML is based on the design by contract paradigm with class invariants and method contracts [46]. Class invariants describe properties that must be preserved by all methods. Method contracts are composed by pre-conditions, post-conditions, and frame conditions, indicating the heap locations which a method may modify. With this, KeY can employ modular verification.

The example in Listing 6 uses a specification technique based on **dynamic frames** [40] and is adapted from the *Cell* example by Smans et al. [58]. The cell's specification starts by declaring a set of locations called `footprint` (line 4) composed only by the `x` field (line 6). The `accessible` annotation (line 5) specifies that the set `footprint` will only change if a location in the set mentioned on the right-side of the colon changes. In this case, `footprint` is constant. Then an invariant is stated (line 9) and it is specified that it only depends on locations in `footprint`. The `setX` method will not throw exceptions (indicated with the `normal_behavior` annotation in line 12), it assigns only to locations in the `footprint` (line 13), requires the parameter to be positive (line 14), and ensures the next call to `getX` will return the given value (line 15). Since it does not perform reads from heap locations, the `accessible` annotation is omitted.

Listing 6 JML specification example.

```
1  class Cell {
2    private int x;
3
4    /*@ model \locset footprint;
5      @ accessible footprint: \nothing;
6      @ represents footprint = x;
7      @*/
8
9    //@ invariant x > 0;
10   //@ accessible \inv: footprint;
11
12   /*@ normal_behavior
13     @ assignable footprint;
14     @ requires value > 0;
15     @ ensures getX() == value;
16     @*/
```

```
17    void setX(int value) {
18      x = value;
19    }
20  }
```

As far as we know, KeY is the verification tool that supports the greatest number of Java features among the other static verification tools for Java, allowing one to verify real programs considering the actual Java runtime semantics. This includes reasoning about inheritance, dynamic method lookup, runtime exceptions, and static initialization. A consequence of this is that, as the members of the KeY team point out, KeY is not overly suitable for the verification of algorithms that require abstracting away from the code since KeY's main goal is the verification of Java programs [20].

KeY's is based on an interactive theorem prover for first-order Java dynamic logic (JavaDL) [27], which can be seen as a generalization of Hoare logic [30]. An important part in the construction of proofs in KeY is symbolic execution. This process takes every possible execution branch and transforms the program leading to a set of constraints, which can then be verified against the specification. KeY provides a semi-automated environment where the user may choose to apply every step of the proof, apply a strategy macro, combining several deductive steps, or execute an automated proof search strategy. As well as offering a high degree of automation, KeY supports SMT solvers, such as Z3 [23], which are often useful to solve arithmetical problems [20]. More details on how to use KeY may be found online.[3]

The most common strategy macros, which we have significantly used in our experiments, are: propositional expansion (to apply propositional rules); finish symbolic execution (to apply only rules for modal operators of dynamic logic, thus executing Java programs symbolically); close provable goals (automatically close all goals that are provable, but do not apply any rules to goals which cannot be closed).

3 Experiments

In this section, we start by giving an overview of the examples that are going to be used, showing the object usage protocols and their implementations, as well as, client code using the presented objects (in Section 3.1). The code is in Java, a language supported by all the aforementioned tools, which given its object-oriented nature, is well-suited for building objects with protocol where method calls act like transitions of a state-machine.

Then in Sections 3.2 and 3.3, the common Java code for the classes is annotated with the appropriate specification for each tool capturing the intended object protocols.

Finally, in Section 3.4, we present annotated client code which uses the object code defined before, and answer the **RQ**: can the tools check protocol compliance and completion, even when objects are shared in a collection?

All code implemented is available online. Throughout the text, hyperlinks in blue point directly to the lines of code relevant to what is being discussed in order to help the reader and to avoid the need to download code. Nonetheless, crucial code parts are presented in listings and discussed in detail. Additionally, a thorough discussion of the implementation is also online for the interested reader.

[3] https://www.key-project.org/docs/UsingKeyBook/

3.1 Running examples

In this study, the main objects with protocol we consider are file readers. Their usage protocol is shown in Figure 2. Circles represent states, arrows denote transitions performed by method calls, and diamonds represent a decision based on the return value of the preceding call. The initial state is marked with an incoming arrow without an outgoing state. The final state is marked with a thicker border. We refer the reader to [60, 61] for more information about this kind of automata, called *Deterministic Object Automata*, and a tool generating those.

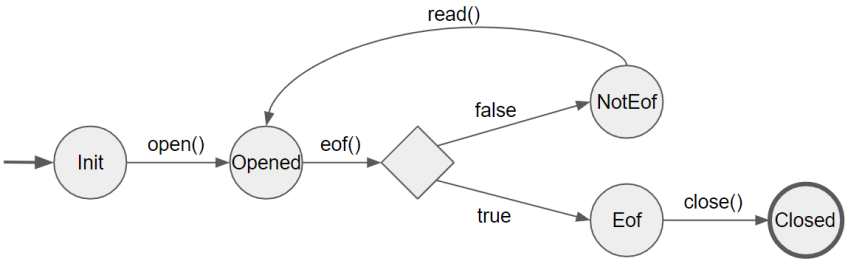

Figure 2 File reader's protocol.

According to the file reader's protocol, the `open` method must be initially called. Then, one must call the `eof` method to check if the end of the file was reached. While it returns `false`, `read` calls are allowed. Otherwise, the whole file was read and the `close` method needs to be called to terminate the protocol. A usage example of a file reader exhibiting protocol compliance and completion is shown in Listing 7.

Listing 7 FileReader's usage example.

```
1 FileReader f = new FileReader("file.txt");
2 f.open();
3 while (!f.eof()) {
4   f.read();
5 }
6 f.close();
```

To track the number of bytes still available to be read, the reader has an internal `remaining` field. Additionally, we may use a `state` field to track the current state of the object. This may be seen as unnatural and superfluous, but when there is no primitive notion of typestates in the language, it is unavoidable. One example of this encoding of protocols being employed is in the Casino contract presented in the VerifyThis event.[4] Exemplifying code is presented in Listing 8. Assume that methods prefixed with `file_` perform actions on the file system. Please note that in the tool specific implementations, files are not actually read, so as to simplify the code for demonstration purposes.

Listing 8 FileReader's code.

```
1 class FileReader {
2   String filename; State state; int remaining;
3
4   FileReader(String name) {
5     filename = name;
6     state = INIT;
7   }
```

[4] https://verifythis.github.io/casino/

```
8
9     void open() {
10      assert (state == INIT);
11      remaining = file_size(filename);
12      state = OPENED;
13    }
14
15    boolean eof() {
16      assert (state == OPENED);
17      return remaining == 0;
18    }
19
20    byte read() {
21      assert (state == OPENED && remaining > 0);
22      remaining--;
23      return file_byte_read(filename);
24    }
25
26    void close() {
27      assert (state == OPENED && remaining == 0);
28      file_close(filename);
29      state = CLOSED;
30    }
31  }
```

After implementing and specifying the file readers, we implement a collection to store these. The challenge is then to verify the collection and statically track the different states of the stored file readers, while ensuring protocol compliance and completion.

For our collection, we implement a singly-linked-list, meaning that each node has a reference only to the next node. This list has two fields: `head` and `tail`. The former points to the first node, the latter points to the last node, as it is commonly implemented in imperative languages. Items are added to the `tail` and removed from the `head`, following a FIFO discipline. Having a `tail` field is crucial for efficiency, avoiding the need to iterate all the nodes before adding a new node to the end. Code for it is presented in Listing 9.

Listing 9 Linked-list's code.

```
1   class Node<T> {
2     T value; Node next = null; Node(T v) { value = v; }
3   }
4
5   class LinkedList<T> {
6     Node<T> head = null; Node<T> tail = null;
7
8     void add(T value) {
9       if (head == null) {
10        head = tail = new Node(value);
11      } else {
12        tail.next = new Node(value);
13        tail = tail.next;
14      }
15    }
16
17    T remove() {
18      assert (head != null);
19      T value = head.value;
20      if (head == tail) {
21        head = tail = null;
22      } else {
23        head = head.next;
24      }
25      return value;
26    }
```

```
27
28    boolean isEmpty() { return head == null; }
29
30    LinkedListIterator<T> iterator() {
31      return new LinkedListIterator(head);
32    }
33  }
```

We believe implementing a linked-list is particularly relevant for a number of reasons. Besides being a very common data structure, and quite simple in nature, its use of pointers often creates challenges for type systems without support for *deductive reasoning*. For example, in a type system with a strict ownership discipline, it is difficult to deal with the aliasing between the `tail` field and the second to last node's `next` field.[5] Additionally, matching the concrete structure of the collection (i.e. the linked nodes) with the abstract representation, to be able to track the states of the stored objects, usually requires ghost code and (again) deductive reasoning, as we will observe.

Although the number of values stored in the linked-list is arbitrary, we can define a finite protocol (Figure 3) that over-approximates the possible states: the list may be empty, which means that we are only allowed to add new values; or the list may be not empty, which means that we can also remove at least one value.[6] With just these two states it is unknown if the list becomes empty or not after removing a value, so we need to encode this uncertainty with an additional state and use the `isEmpty` method to check the emptiness of the list.

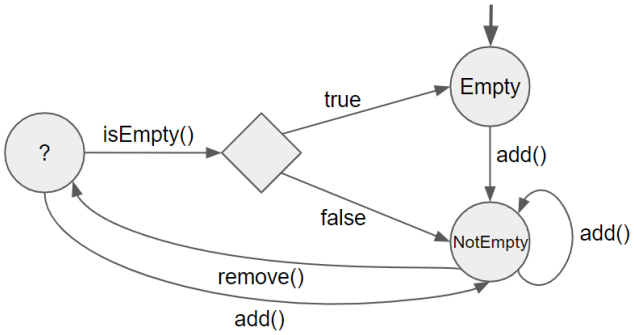

Figure 3 List's protocol.

Similarly, we can define a protocol for an iterator over the linked-list (Figure 4) with three states: one indicating that there are values to retrieve with the `next` method, another to specify that we reached the end of the list, and finally, a state to encode the uncertainty between the previous two, where the `hasNext` method may be used to check if there are still values to return. Code for the iterator is presented in Listing 10.

Listing 10 Iterator's code.

```
1  class LinkedListIterator<T> {
2    Node<T> curr;
3    LinkedListIterator(Node head) { curr = head; }
```

[5] For instance, in Rust, the ownership discipline prevents one from creating linked-lists, unless `unsafe` code is used. GhostCell, a recent solution to deal with this, allows for internal sharing but the collection itself still has to respect the discipline [62]. It uses `unsafe` code for its implementation but was proven safe with separation logic.

[6] Context-free session types can be used to describe protocols which are not limited by the expressiveness of regular languages [2].

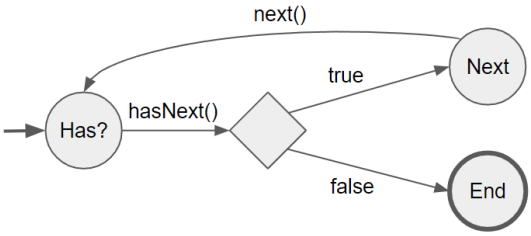

Figure 4 Iterator's protocol.

```
4
5     boolean hasNext() {
6       return curr != null;
7     }
8
9     T next() {
10      assert (curr != null);
11      T value = curr.value;
12      curr = curr.next;
13      return value;
14    }
15  }
```

An example combining the three classes is shown in Listing 11. In this example, a number of file readers in their initial state is added to the list (line 6). This collection is then passed to the auxiliary method `useFiles` (line 7) which iterates through all the readers and, for each one, follows their protocol to the end. The properties of protocol compliance and completion hold for this program example.

Listing 11 Usage example in code.

```
1  void main() {
2    LinkedList list = new LinkedList();
3    FileReader f1 = new FileReader("a");
4    FileReader f2 = new FileReader("b");
5    FileReader f3 = new FileReader("c");
6    list.add(f1); list.add(f2); list.add(f3);
7    useFiles(list);
8  }
9
10 void useFiles(LinkedList<FileReader> list) {
11   LinkedListIterator it = list.iterator();
12   while (it.hasNext()) {
13     FileReader f = it.next();
14     f.open();
15     while (!f.eof()) f.read();
16     f.close();
17   }
18 }
```

We will now report on our experience with each tool detailing the specification and proof effort required, keeping the issues of protocol compliance and completion in mind. In summary, we successfully verify the file reader class (Listing 8) and its use (Listing 7) in all four tools. The linked-list and iterator implementations (Listing 9 and 10), as well as their usages (Listing 11), are accepted by all except Plural. Protocol completion can be guaranteed with workarounds in all tools except Plural.

3.2 File reader specification

VeriFast

To implement the file reader, we use pre- and post-conditions in all public methods indicating the required and the ensured states after the call (example in List. 12). The fields of this object are `state` (to keep track of the current state), and `remaining` (to keep track of the number of bytes still to read). Every time the state needs to change, we assign to the `state` field. We use constants to identify different states, thus avoiding the use of literal numbers in specifications. To request access to the fields of the object, and to enforce that `remaining` is equal to or greater than zero, we define the `filereader` predicate.

Listing 12 `close` method in VeriFast.

```
1 public void close()
2   //@ requires filereader(this, STATE_OPENED, 0);
3   //@ ensures filereader(this, STATE_CLOSED, 0);
4 {
5   this.state = STATE_CLOSED;
6 }
```

VerCors

The implementation in VerCors is very similar to the one in VeriFast, except for these differences: we do not define a predicate to abstract the contents of the object (instead we keep access to the fields exposed for practical reasons); and since VerCors supports ghost fields, we use one to track the state, using numbers to represent each state.

Plural

Given the support for typestates, we directly define three states, *init*, *opened*, and *closed*, which refine (i.e. define a substate of) the root state `alive`, a state in which all objects are in. Additionally, we define two states, *eof* and *notEof*, refining *opened*, indicating if we have reached or not the end of the file, respectively. The file reader has a boolean field, `remaining`, indicating if there is something to read. To enforce the relation between the states *eof* and *notEof*, and the `remaining` field, we define invariants for these states. Due to a limitation, we had to simplify the `read` method, making it read the file all at once.[7] To enforce the protocol, we declare for each method the required and ensured states as well as the permissions needed to perform each call. The `open`, `read`, and `close` methods require `unique` permission to the receiver object. *Full* permissions would be enough except for the possibility of concurrent accesses, which would require methods to be `synchronized`. The `eof` method only needs an *immutable* permission guaranteeing that we are in the `opened` state, and returns a boolean value indicating if the end of the file was reached.

KeY

We also model the protocol with pre- and post-conditions in methods. The state is tracked using an integer ghost field. As in VeriFast and VerCors, `remaining` stores the number of bytes left to read, and we enforce the value to be equal or greater than zero with an

[7] Ideally, `remaining` would store an integer, but the syntax `remaining == 0` does not seem to be supported in the invariants. In consequence, we cannot model the arbitrary number of bytes to read.

invariant. The file reader has a footprint, composed by the `state` and `remaining` fields, and each of its methods specifies which fields may be modified, according to the dynamic frames technique [40]. To verify this class, we have to prove each method correct, according to each specification, and the fact that nothing changes the footprint. Given its simplicity, all proofs were automatically done using KeY's default strategy.

Evaluation

As expected, we successfully modelled a protocol for a file reader in all four tools: in Plural, the implementation was mostly straightforward given the support for typestates, but annotations were required in all methods; in VeriFast, VerCors, and KeY, we used method contracts, which also required some annotation burden. Thus, we motivate the need for more natural ways to specify protocols, for example, via automata, which helps the programmer visualize and design the protocol.

3.3 Linked-list and iterator specifications

VeriFast

The linked-list implementation is adapted and extended from a C implementation available online. 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. This matches common implementations and makes verification more challenging because we have to avoid null pointer errors.

To model the structure of the list, we define a predicate that holds access to the `head` and `tail` fields and of all the nodes in the list (List. 13). The only input parameter is the reference to the linked-list. The output parameters are the references to the head and tail, and a ghost list to reason about the values in the list in an abstract way (line 1). This ghost collection will be crucial to track the different states of the file readers stored in the list. 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, we request access to the sequence of nodes between the head (inclusive) and the tail (exclusive), through the `lseg` predicate, and then keep access to the tail node separately (line 5). The `node` predicate holds the permissions to the `next` and `value` fields of a given node. Note that we do not hold permission to the fields of the values stored. This is to allow them to change independently of the linked-list.

Listing 13 `llist` predicate in VeriFast.

```
1 predicate llist(LinkedList obj; Node h, Node t, list<FileReader> list) =
2   obj.head |-> h &*& obj.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 implementation of the remove and isEmpty methods is straightforward requiring only the unfolding and folding of the `llist` and `lseg` predicates a few times. The add method requires an auxiliary lemma (List. 14) 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 14 `add_lemma` lemma in VeriFast.

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

To implement the iterator, we define a `iterator` predicate which holds access to the current node field and all the nodes in the linked-list. Then, we split the permissions to the nodes in two parts (List. 15): 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); and a sequence from the current node (inclusive) to the final one. Both parts allow us to reason on the values already seen, and the values still to be seen. This split occurs when the iterator is created. After iterating all the nodes, the full permission to the nodes needs to be restored to the list, which is done via an auxiliary lemma.

Listing 15 `iterator_base` predicate in VeriFast.

```
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);
```

The implementation of the hasNext method is straightforward. The implementation of the next method requires unfolding and folding predicates, the use of a lemma showing 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, moving the just retrieved value from the "to see" list to the "seen" list (List. 16).

Listing 16 `iterator_advance` lemma in VeriFast.

```
1 lemma void iterator_advance(Node h, Node n, Node t)
2   requires [1/2]lseg(h, n, ?a) &*& [1/2]node(n, ?next, ?val1) &*&
3     [1/2]nodes(next, ?b) &*& [1/2]lseg(h, t, ?list) &*&
4       [1/2]node(t, null, ?val2);
5   ensures [1/2]lseg(h, next, append(a, cons(val1, nil))) &*&
6     [1/2]nodes(next, b) &*& [1/2]lseg(h, t, list) &*&
7       [1/2]node(t, null, val2);
```

VerCors

The implementations of the linked-list and iterator closely follow the VeriFast's ones, with just some differences. Given that predicates in VerCors do not support output parameters, we have predicates to request access to the needed memory locations, and then methods to build the ghost lists that allow us to track the values. Additionally, we use `given` and `yields` clauses in the methods to receive and return the necessary lists (lines 1-2 of List. 17). Instead of using these clauses in methods, we would have preferred to rely on ghost fields storing those lists. Unfortunately, it seems one cannot reason about the old value of a field if its permission is inside a predicate. Using inline unfolding does not seem to work.

Listing 17 `remove` method's contract in VerCors.

```
1 given seq<FileReader> oldList;
2 yields seq<FileReader> newList;
3 requires state(oldList) ** |oldList| > 0;
4 ensures state(newList) ** newList == tail(oldList);
```

Linked-lists and iterators have been implemented before for VerCors [14] but, to our knowledge, there is no list implementation that uses a `tail` field, preferring instead a recursive approach, with the "head" being the first value, and the "tail" being the rest of the list.

Plural

For the linked-list, we adapt a stack example from Plural's repository.[8] Naturally, we make the appropriate changes since our linked-list follows a FIFO discipline, while a stack follows a LIFO one. Since objects in Plural should be associated with typestates, both our `Node` and `LinkedList` classes have protocols.

In the `Node` class, we define two orthogonal state dimensions, *dimValue* and *dimNext*, which handle the `value` and `next` fields, respectively. In *dimValue* there are two states, *withValue* and *withoutValue*, which indicate if the node has permission to the stored value or not. In *dimNext* we have states *withNext* and *withoutNext*, which say if the node has permission to the next node or if `next` is `null`. State dimensions avoid the need to reason about all the combinations of having (or not) a value and having (or not) a next node. Since direct field accesses are disallowed, we define getter and setter methods for both fields.

In the `LinkedList` class, we define two states: the empty and the non-empty. When it is empty, the `head` and `tail` fields are `null`. When it is not empty, `head` and *tail* are not `null` and there is unique permission to the first node, which is pointed by `head`. Since the `head` points to the next node, and so on, we should have the required chain of nodes that builds the linked-list. Adding and removing values from the list require *unique* permission to it.

Unfortunately, Plural did not accept either implementation. With regards to the `Node` class, we had errors in all the methods indicating that the receiver could not be packed (i.e. coerce from the concrete field view of the class to the abstract typestate view [24]) to match the state specified by the `ensures` annotation parameter. Additionally, the invariant for the *withValue* state had an error stating that the parametric permission kind we specified was unknown, even though that was introduced with the appropriate annotation. In fact, we did the same for the `LinkedList` class and we did not get these kinds of errors.

With regards to `LinkedList`, the only errors reported were in the `add` method. To understand why, consider the case in which the list is not empty. In this case, the `tail` is non-null, and we must call `setNext` on it to append a new node (line 8 of List. 18). However, we do not have permission to do that. For this to work, we would need to have permission to the last node that is owned by the second to last node. Unfortunately, we could not perform such transferring of permissions. An alternative solution could be to use *share* permissions instead of *unique* ones in the nodes. But 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 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. Given this, we did not attempt to implement the iterator.

Listing 18 LinkedList's `add` method in Plural.

```
1  @Unique(requires="alive", ensures="notEmpty", use=Use.FIELDS)
2  public void add(@PolyVar(value="p", returned=false) T value) {
3    @Apply("p") Node<T> n = new Node<T>(value);
4    if (head == null) {
5      head = n;
6      tail = n;
7    } else {
8      tail.setNext(n);
9      tail = n;
10   }
11 }
```

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

KeY

The linked-list implementation is heavily inspired in a tutorial by Hiep et al. which implements a doubly-linked-list [29]. We declare several fields: `head` and `tail`; `size`, to count the number of values; `nodeList`, containing a sequence of nodes; and `values`, containing a sequence of values. The `nodeList` and `values` fields are ghost fields. As for the file reader, we define the linked-list's footprint, composed by its fields and the fields of all the nodes (line 5 of List. 19). We also specify that the footprint itself only changes if the nodes sequence changes (line 2), and that the list's invariant only depends on the locations in the footprint (line 3). The proof of the former was generated automatically by KeY using the default strategy. The proof of the latter required some interactivity to guide the proof. Note that the footprints of the values are not part of list's footprint.

Listing 19 List's footprint in KeY.

```
1 public model \locset footprint;
2 accessible footprint: nodeList;
3 accessible \inv: footprint;
4 represents footprint = size, head, tail, nodeList, values,
5   (\infinite_union \bigint i; 0 <= i < nodeList.length; ((Node)nodeList[i
      ]).*);
```

The list's invariant is the most verbose part of the specification. First, we specify that `size` is equal to the number of nodes, which is then equal to the number of values. Then we enforce that the values in the list are not `null`. We also use an existential quantifier, indicating that in each position of the sequences, elements exist (lines 1-2 of List. 20). This is necessary because KeY treats sequences in a way where they may occasionally contain not-yet-created objects. Additionally, since the sequence declarations do not enforce the type of their elements, we need to do it explicitly, either using an `instanceof` operator, or using an existential quantifier. Furthermore, we need to cast the result of accessing a position in a given sequence. Following that, we have to take into account that the list may be empty. So, we define that either the nodes sequence is empty, and the `head` and `tail` fields are `null`, or the nodes sequence is not empty, and the `head` points to the first node, and `tail` points to the last one (lines 3-6). To ensure we have a linked-list, we enforce that the `next` field of each node points to the following node in the sequence (lines 7-8). We also enforce that all the nodes are distinct (lines 9-12). Finally, we specify that each value in the values sequence corresponds to the value stored in each node in the same position.

Listing 20 List's invariant in KeY.

```
1  (\forall \bigint i; 0 <= i < values.length;
2      (\exists FileReader f; f == values[i] && f != null)) &&
3  ((nodeList == \seq_empty && head == null && tail == null)
4    || (nodeList != \seq_empty && head != null && tail != null &&
5      tail.next == null && head == (Node)nodeList[0] &&
6      tail == (Node)nodeList[nodeList.length-1])) &&
7  (\forall \bigint i; 0 <= i < nodeList.length-1;
8      ((Node)nodeList[i]).next == (Node)nodeList[i+1]) &&
9  (\forall \bigint i; 0 <= i < nodeList.length;
10   (\forall \bigint j; 0 <= j < nodeList.length;
11     (Node)nodeList[i] == (Node)nodeList[j] ==> i == j
12   )) && ...
```

The iterator has several fields: `list`, a reference to the list; `curr`, the current node; `index`, the position of the current node in the nodes sequence; `seen`, the sequence of values already iterated; and `to_see`, the sequence of values still to be iterated. The `index`, `seen`, and `to_see` fields are ghost fields. As usual, we define the iterator's footprint, composed only

by its fields (line 4 of List. 21). We also specify that the footprint itself does not change (line 2), and that the iterator's invariant depends on its footprint and on the list's footprint (line 3). The proof of the former was done automatically. For the proof of the latter, KeY's default strategy was not enough. The reason for this was that KeY was applying multiple "cut" tactics to try to close the proof for each possible value of `size`. Nonetheless, it was mostly straightforward to guide the proof. We just had to use the "observerDependency" tactic to establish that the iterator's invariant does not change in the presence of heap updates on locations that do not belong to its footprint or the list's footprint.

Listing 21 Iterator's footprint in KeY.

```
1 public model \locset footprint;
2 accessible footprint: \nothing;
3 accessible \inv: footprint, list.footprint;
4 represents footprint = list, curr, index, seen, to_see;
```

In the iterator's invariant we first specify that `index` is a value between zero and the number of values. This number may be equal to the number of values if and only if we have already iterated through all values. Then we enforce that the values in both sequences are not `null`. Following that, we define that the `seen` sequence corresponds to the values already seen, from position zero (inclusive) to `index` (exclusive), and that `to_see` corresponds to the values to see, from position `index` (inclusive) to the end. Since `curr` points to the current node, we map it to position `index` in the nodes sequence, or we specify that it is `null`, when iteration is done. Finally, we assert that the list's invariant holds.

Regarding the linked-list, verifying the constructor, `add`, and `iterator` methods required only the default strategy, but for the `remove` method, some interactivity was need, namely to show that the first value was the value of the `head`. Regarding the iterator, we had to guide the proof of the constructor, mostly to establish the invariants of the iterator and list, since KeY was applying "cut" multiple times, as before. The `hasNext` method was verified automatically with the default strategy. To verify the `next` method, we had to prove that: 1. only the list's and iterator's footprints are accessible; 2. the post-condition holds after execution; 3. and that only the iterator's footprint is modified. These proof requirements required a lot of work, likely because of the relation between `index` and `curr`, which was probably not obvious to the default strategy. Examples of goals which required some effort to prove were: showing that the value of the current node was the first value in the "to see" sequence, proving that such a value was a file reader, and that the new sequences respected the invariant (after the current value was moved to the "seen" sequence).

Evaluation

In VeriFast, as well as in VerCors, the expressiveness of the logic allowed us to specify a linked-list and an iterator. However, deductive reasoning was often required. In our experience, we spent more time in proving results than in writing code, having to unfold and fold predicates very often, revealing their definitions, having to define multiple lemmas, and insert assertions to guide proofs. In VeriFast, we had to write about 160 lines of lemmas. In VerCors, we wrote about 100 lines. Some of the time spent in VerCors with the proofs was reduced because we could reuse the experience we had with VeriFast.

In Plural, we were not able to implement a linked-list. We believe the support for logical predicates would be necessary to be able to specify structures with recursive properties. Furthermore, as far as we can tell, there is no support for parametric typestates, even though there is for fractional permissions, which could potentially allow one to model a list with objects in different states that evolve.

In KeY, we were able to implement and verify a linked-list and an iterator. Although KeY supports interactivity, together with useful macros and a high degree of automation, we spent some time proving properties about the heap. For example, we often had to prove that the footprints of two objects were disjoint, which means we also had to account for possible changes in the footprints themselves, which became very cumbersome. To do this, we had to make the footprints public so that they could be opened in proofs, otherwise we are not sure if we would have been able to finish the proofs. We believe this motivates the need to be able to prove heap and functional properties separately.

Given the differences between the approaches presented, we believe simply comparing lines of specification would not provide a meaningful comparison. Thus, we provide a qualitative evaluation. In summary, we observe that these methods require an important effort especially when one is learning the approaches. Not surprisingly, in VeriFast, VerCors, and KeY, the specification took more space than the code, and was usually verbose. VeriFast and VerCors also required a substantial amount of annotations to guide the proofs. With KeY, the space, usually needed for proofs in the other two tools, was replaced by the time spent proving results interactivity. Even if it turns out that with training, it is not that hard to specify, implement, and verify the examples, it is certainly time consuming. We believe this motivates further study on what we can delegate to static analysis to ease this effort. Plural has less expressive power than the other tools, so it makes sense that the annotation effort was low.

3.4 RQ evaluation

We are able to produce examples of file readers usage, as presented in Section 3.1, where we successfully ensure that the protocol is followed (i.e. only the allowed methods in each state can be called), in all four tools: in VeriFast, VerCors, and KeY, thanks to the pre- and post-conditions; and in Plural, thanks to the typestate abstraction directly supported. We can also ensure protocol compliance when different file readers are stored within a linked-list, in VeriFast (List. 22), VerCors, and KeY, but with significant annotation and proof effort, as observed in the examples produced.

Listing 22 `useFiles` specification in VeriFast.

```
1 requires list != null &*& llist(list, _, _, ?l) &*& tracker(length(l))
      &*& foreachp(l, INV(FileReader.STATE_INIT));
2 ensures list != null &*& llist(list, _, _, l) &*& tracker(0) &*& foreachp
      (l, INV(FileReader.STATE_CLOSED));
```

In the rest of this section, we focus our presentation on the crucial property of **protocol completion**, which guarantees that if and when the program terminates, all the objects are in the final states of their protocols.

VeriFast

To ensure that all file readers created through the lifetime of the program reach the end of their protocol, we define a `tracker` predicate in a *.javaspec* file which keeps hold of the number of open file readers (List. 23). This proposed solution is based on a private exchange with Jacobs [37]. Then, we augment the file reader's specification to increment this counter in the constructor, and decrement the counter in the `close` method. Finally, since we want to guarantee protocol completion for all created objects, we assert in the pre-condition and post-condition of the `main` method that the counter should be zero, given that `main` is the starting (and ending) point of Java programs.

Listing 23 `tracker.javaspec` file.

```
predicate tracker(int count);

lemma void increment_tracker();
  requires tracker(?n);
  ensures tracker(n + 1);

lemma void decrement_tracker();
  requires tracker(?n);
  ensures tracker(n - 1);
```

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 include the post-condition in 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 properties, but if such code is not correctly connected with the "real" code, then the property we desired to establish is not actually guaranteed.

VerCors

To ensure protocol completion, we follow the previous idea, but instead of a "global counter" defined through a predicate written in a specification file, we create a `FileTracker` object which keeps hold of the number of open file readers using ghost code. When we augment the file reader's implementation to increment and decrement the counter in the appropriate methods, we also have to pass the tracker using the `given` directive. Again, it is possible to fall for the same "traps": forgetting to increment and decrement the counter, and forgetting to add the post-condition to the `main` method.

Since VerCors supports quantifiers, one could think of quantifying over all file readers and ensuring they are all closed. Unfortunately, we would be quantifying over all possible file readers, not just the ones actually allocated on the heap.

Plural

Although the typestate abstraction is directly supported, protocol completion is not guaranteed since permissions may be "dropped", as seen in List. 24, where a unique permission for an object is received but not used, without any error being reported. This was not an issue in other tools, even though leaking resources is permitted, because the support for deductive reasoning allowed us to count the number of active objects.

Listing 24 Dropping file reader in Plural.

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

As far as we can tell, Plural's specification language is based on linear logic, which would imply that this would not be permitted. However, we understand why this is the case in Java, since it is common for one 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 important method calls are not omitted and resources are freed (e.g. closing a socket).

KeY

To ensure that all file readers reach the end of their protocol, we define a contract for the `main` method such that for all file readers created at some point in the program, they are in the final state (line 2 of List. 25). Since KeY is not aware of the objects created or not before, we define as pre-condition that no file readers exist when `main` is called (line 1). This works because quantifiers in KeY only reason over objects in the heap. Given that `main` is the first method called in a Java program, this requirement is actually an assumption.

Listing 25 `main` method's contract with protocol completion in KeY.

```
1 requires !(\exists FileReader f; true);
2 ensures (\forall FileReader f; f.state == FileReader.STATE_CLOSED);
```

Thanks to first-order dynamic logic, we can use quantifiers to specify that all existing file readers should have their protocol completed. Although this seems powerful, we have to adapt all other methods to specify that no new file readers are created inside. This is necessary because it would be possible for methods to create file readers only available in the scope of their execution, which would exist in the heap as created objects, but for which we would know nothing about. This post-condition, written as `!(\exists FileReader f; \fresh(f))`, was added in all needed methods. In the file reader's constructor, we also had to say that `f` was different from `this`, since the newly created reference is fresh.

Evaluation

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 any of the logics employed by all four tools. 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.

One workaround we found for VeriFast and VerCors was to have a counter that keeps track of all typestated-objects which are not in the final state. This requires keeping hold of the aforementioned tracker in specifications, which can be a huge burden in bigger programs. Ensuring protocol completion could be embedded in separation logic and such a feature could even be possible in VeriFast. Given that VeriFast supports leak checking, one would just need to incorporate notions of typestates and ensure that leaking would only be 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.

For Plural, we did not find a way to ensure protocol completion. This could be supported by asking the programmer to indicate which state of a given object is the final one and only allowing permissions for *ended* objects to be "dropped".

In KeY, we made use of the support for universal quantifiers and reasoning on heap-dependent expressions. However, verifying the code against that specification can be cumbersome, and requires augmenting the specifications of all other methods, ensuring no untracked objects are added to the heap.

4 General assessment of the tools

In this section, we summarize our views about the tools, using the knowledge gained from our experiments, and provide suggestions of what could be improved.

VeriFast

Separation logic, fractional permissions, and predicates, allow 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, as we have seen when implementing the linked-list (Section 3.3). This is tedious and can be a barrier to less experienced users. Although VeriFast's IDE provides a way for one to look at each step of a proof when an error is discovered, we believe that, at least in part, having a way to guide proofs (like one can do with proof assistants such as Coq), would improve the user experience even more by: (1) avoiding having to insert proof guiding assertions in the code implementation itself (allowing for more separation of concerns); and (2) avoiding the need to rerun the tool every time that occurs. In other words, when guiding the proofs, one would get immediate feedback. Nonetheless, the IDE experience was still very useful in helping us prove several results.

Checking for protocol completion is crucial to ensure that essential method calls are not omitted and that resources are freed. Unfortunately, that concept is not built-in in separation logic (Section 3.4). Given that VeriFast supports leak checking, one would just need to ensure that leaking would only be allowed when objects are in their final states.

VerCors

As in VeriFast, rich and expressive specifications are supported, but deductive reasoning is (again) required. As we noted before, this can be tedious, as highlighted by the time and lines of code we needed to prove results. Unfortunately, VerCors has no interactive experience so we had to practice "trial and error" more often, guessing what could be wrong and rerunning the tool every time we changed the code.

In terms of user experience, we think 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 [31]. We also believe the tool could be more efficient: since specifications are self-framing (i.e. only depend on memory locations that they themselves require to be accessible) and the checking process is modular, VerCors could cache some results to avoid re-checking parts of the code that were not modified. We also noticed that if we unfolded a predicate on which some truth depends on, that knowledge would be lost. For example, the knowledge of the values stored in a sequence of nodes depends on the permissions to those nodes, available in the `nodes_until` predicate. In principle, unfolding this predicate should not invalidate the available information, but it does. 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.

Comparison between VeriFast and VerCors

Given the similarities between VeriFast and VerCors, we believe it is very relevant to provide a comparison between both.

With respect to specifying access to memory locations, VeriFast only supports the points-to assertions of separation logic, while VerCors also supports permission annotations, inspired by Chalice [44, 45], allowing us to refer to values in variables without the need to use new names for them, which was very useful when writing the specifications. Furthermore, VerCors has built-in support for quantifiers, many different abstract data structures, and ghost code, which VeriFast does not. We used a fair amount of ghost code in VerCors. Nonetheless, VeriFast supports the definition of new inductive data types, fixpoint functions, higher-order predicates, and counting permissions, which VerCors does not. Unfortunately, VerCors does

not support generics in Java. Regarding VeriFast, as Bart Jacobs points out, at the time of writing, "support for Java generics in VeriFast is in its infancy".

Both provide support for fractional permissions, which we have used. However, 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 be avoided as much as possible. VerCors lacks support for counting permissions, which would allow permissions to be split in other ways.

Finally, we missed the support for output parameters which VeriFast has. To reproduce the same concept in VerCors, 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.

Plural

The rich set of access permissions allows objects' state to be tracked even in the presence of aliasing, and permits read/write and write/write operations, thanks to state guarantees. Nonetheless, we could not specify structures such as the linked-list with double handle (i.e. with head and tail fields), likely because of the lack of support for logical predicates. Furthermore, to our knowledge, there is no support for parametric typestates.

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. 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 [7]. But as far as we know, the idea was not realized.

Furthermore, there is no built-in guarantee of protocol completion. This could be provided by only permitting permissions for *ended* objects to be "dropped" (Section 3.4).

KeY

The use of JML for specifications, a language for formally specifying behavior of Java code, used by various tools, reduces the learning curve for those that already know JML. Furthermore, KeY supports a great number of Java features, allowing one to verify real programs considering the actual Java runtime semantics. Nonetheless, generics are not supported, although there is an automated tool to remove generics from Java programs, which can then be verified with KeY. Because of its focus on Java, KeY is not overly suitable for the verification of algorithms that require abstracting away from the code [20].

One important aspect that makes KeY stand out from other tools is the support for interactivity, which allows the programmer to guide the proofs. This is an aspect that we missed when experimenting with other tools. Additionally, KeY provides useful macros and a high degree of automation, as well as support for SMT solvers, such as Z3. We used Z3 often to more quickly close provable goals, specially those involving universal quantifiers.

Since KeY's core is based on first-order dynamic logic, one can express heap-dependent expressions: the heap is an explicitly object in the logic. Although this allows for much expressiveness, it often becomes very difficult to verify programs, as our experience has shown

(Section 3.3). We think that a variant of KeY that would instead use separation logic, to abstract away the heaps and the notion of disjointness, would be very helpful, improving readability and reducing verbosity.

Nonetheless, there are probably other alternatives to separation logic that would help in solving the aforementioned issues. In a private conversation with KeY's group leaders, they point out that the connectives of separation logic "would get in the way of automation", a crucial feature of KeY. For example, it seems that "the heap separation rule tends to split proofs too early" [35]. Furthermore, they mention that the problems we encountered could be summarized in two main points: (1) insufficient abstract specification primitives; (2) inability to prove heap and functional properties separately, in a modular fashion. KeY's team is aware of these issues and will address them in the future (at the time of writing).

As we pointed out above, we believe separation logic together with resource leaking prevention (except for objects with completed protocol), could be used to ensure protocol completion without the need for adding extraneous specifications. This would be another reason why we believe separation logic would be preferable over first-order dynamic logic, but it is possible there are other alternatives.

Finally, we enjoyed the user experience and appreciated that KeY comes with examples to experiment with. Nonetheless, at the moment of writing, we believe there is room for improvement. For more details, we refer the reader to a thorough discussion of the issues we found and suggestions for improvements, which we shared with KeY's team.

5 Related work

Penninckx et al. develop an approach to verify input/output properties of programs [54]. They encode I/O behavior using abstract permission-based predicates implemented in VeriFast. The technique ensures that a program only performs the allowed I/O operations. Additionally, it guarantees a terminated program has performed all desired operations with a post-condition specifying the final state a program should be found in. Later, Jacobs presented an approach to verify liveness properties [36]. Blom et al. verify the functional behavior of concurrent software using histories, which record the actions taken by a concurrent program [15]. The technique has been integrated in VerCors and experimentally added to VeriFast. Similarly, Oortwijn integrated process algebra models [8] in VerCors to reason about functional properties of shared-memory concurrent programs, including non-terminating ones [52]. More recently, work has been developed to support the deductive verification of JavaBIP models in VerCors [12]. In these models, Java classes are considered as components where their behavior is described by finite state machines, and component interactions are specified with synchronization annotations [11]. Kim et al. propose a technique to specify protocols of Java classes by incorporating typestates into JML [41]. When translating their extension to pure JML, multiple boolean fields for each state are declared which, when `true`, indicate the object is in that given state. Multiple fields are needed to support *state refinements* [9]. Cheon and Perumandla extend JML with a new specification clause containing a regular expression-like notation to specify the sequences of method calls allowed for a given class [21].

In this study, we focus on *sequential* examples and only present a simple protocol, which does not require a complex encoding, so the aforementioned techniques would either not be applicable or would introduce unnecessary verification overhead.

With respect to comparison studies, there are several that have been conducted. However, as far as we know, no study was previously done that focused on the verification of *protocol compliance and completion*. Nonetheless, we reference some works we found relevant to

us. Lathouwers and Huisman examine the annotation effort in several tools, including VeriFast and VerCors [42]. Hollander briefly discusses the differences between VeriFast and VerCors [31]. Boerman et al. study the way in which KeY and OpenJML treat JML specifications differently and the effort in switching between both tools [16].

6 Conclusions

In this paper, we address the **RQ** (Page 2) by reviewing four verification tools for Java. In particular, we evaluated their ability to check the correct use of objects with protocol, and if they were able to guarantee protocol completion, even when these were shared in collections (Section 3.4). Additionally, we evaluated the programmer's effort in making the code acceptable to each, and provide suggestions for improvements (Section 4).

We were able to reach a general conclusion: stateful objects usually have protocols representing their intended usage. In the tools we have studied, protocols are not first-class entities; instead they need to be encoded with method contracts (even in Plural, annotations on methods are required). In contrast, approaches based on behavioral types, where protocols can be defined via automata [4, 47, 60], treat protocols as central, and provide a global view of the intended usage of each object. By having this model, reasoning on relevant properties becomes easier than on lower level encodings.

Now we summarize key points gathered from our experiments with each tool. Both VeriFast and VerCors support rich and expressive specifications based on separation logic: this allowed us to successfully address the **RQ**. However, deductive reasoning was often required. This is very demanding and can be a barrier to less experienced users. We believe improved interactive experiences for programmers are key to make these tools more approachable. Furthermore, fractional permissions only allow for read-only access when data is shared.

Plural is different from these tools in two major ways: it does not support logical predicates and so, specifications are less expressive in that regard, which prevented us from implementing a linked-list. Nevertheless, access permissions support more kinds of sharing, but access to thread-local shared data might require an unnatural use of locks.

KeY supports interactivity, automation, and the ability to reuse proofs. Nonetheless, the fact that heaps are mentioned explicitly in assertions made it difficult to read the hypothesis and proof goals. Additionally, we often had to show that certain footprints were disjoint. So, although we successfully answered the **RQ**, again the effort was substantial. To fully automate some proofs, KeY depends on finding the right specifications and proof search settings, which is not easy. More abstract specification primitives, and the ability to separate proofs of heap and functional properties, are crucial features to improve both readability and ease of proving results.

So, we proved protocol compliance with some effort, but protocol completion, crucial to ensure that necessary method calls are not forgotten and that resources are freed, is not directly supported by any of these tools. Although there are workarounds in some, we believe such guarantee should be supplied directly. This could be done by ensuring that no permission to an object is "dropped" unless it is in the final state.

In conclusion, this study motivates the need for lightweight methods to statically guarantee protocol compliance and completion in the presence of several patterns of sharing, like objects with protocol stored in collections, including the following features: usage protocols as the central entity defining objects' behavior, more kinds of sharing beyond fractional permissions also avoiding the need for locks in sequential code, and better techniques to reason about permissions to heap locations.

For completion, this study could be complemented with OpenJML [22], and LiquidJava, a recent tool that integrates liquid types in Java [26].

References

1. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book – From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.
2. Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic lambda calculus with context-free session types. *Inf. Comput.*, 289(Part A), 2022. doi:10.1016/j.ic.2022.104948.
3. Davide Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.
4. Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. A Java typestate checker supporting inheritance. *Sci. Comput. Program.*, 221, 2022. doi:10.1016/j.scico.2022.102844.
5. Bernhard Beckert and Reiner Hähnle. Reasoning and Verification: State of the Art and Current Trends. *IEEE Intell. Syst.*, 29(1):20–29, 2014. doi:10.1109/MIS.2014.3.
6. Nels E. Beckman. Modular typestate checking in concurrent Java programs. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 737–738. ACM, 2009. doi:10.1145/1639950.1639990.
7. 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*, pages 227–244. ACM, 2008. doi:10.1145/1449764.1449783.
8. Jan A. Bergstra and Jan Willem Klop. Process Algebra for Synchronous Communication. *Inf. Control.*, 60(1-3):109–137, 1984. doi:10.1016/S0019-9958(84)80025-X.
9. Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 217–226. ACM, 2005. doi:10.1145/1081706.1081741.
10. 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*, pages 301–320. ACM, 2007. doi:10.1145/1297027.1297050.
11. Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Exogenous coordination of concurrent software components with JavaBIP. *Softw. Pract. Exp.*, 47(11):1801–1836, 2017. doi:10.1002/spe.2495.
12. Simon Bliudze, Petra van Den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java. In *26th International Conference on Fundamental Approaches to Software Engineering*, 2023.
13. Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Proceedings of Integrated Formal Methods*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017. doi:10.1007/978-3-319-66845-1_7.
14. Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. *Int. J. Softw. Tools Technol. Transf.*, 17(6):757–781, 2015. doi:10.1007/s10009-015-0372-3.
15. Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. History-Based Verification of Functional Behaviour of Concurrent Programs. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods – 13th International Conference, Proceedings*, volume 9276 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2015. doi:10.1007/978-3-319-22969-0_6.
16. Jan Boerman, Marieke Huisman, and Sebastiaan J. C. Joosten. Reasoning About JML: Differences Between KeY and OpenJML. In *Integrated Formal Methods – 14th International Conference, Proceedings*, volume 11023 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2018. doi:10.1007/978-3-319-98938-9_3.

17 Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *The 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, 2005. doi:10.1145/1040305.1040327.

18 John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.

19 Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias Jakobsen, Mikkel Kettunen, and António Ravara. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Asian Symposium on Programming Languages and Systems*, volume 12470 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2020. doi:10.1007/978-3-030-64437-6_6.

20 Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with KeY. *Int. J. Softw. Tools Technol. Transf.*, 17(6):729–744, 2015. doi:10.1007/s10009-013-0293-y.

21 Yoonsik Cheon and Ashaveena Perumandla. Specifying and Checking Method Call Sequences in JML. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice*, volume 2, pages 511–516. CSREA Press, 2005.

22 David R. Cok. JML and OpenJML for Java 16. In *FTfJP 2021: 23rd ACM International Workshop on Formal Techniques for Java-like Programs, 2021, Proceedings*, pages 65–67. ACM, 2021. doi:10.1145/3464971.3468417.

23 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

24 Robert DeLine and Manuel Fähndrich. Typestates for Objects. In *18th European Conference on Object-Oriented Programming, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004. doi:10.1007/978-3-540-24851-4_21.

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

26 Catarina Gamboa, Paulo Alexandre Santos, Christopher Steven Timperley, and Alcides Fonseca. User-driven Design and Evaluation of Liquid Types in Java. *CoRR*, abs/2110.05444, 2021. arXiv:2110.05444.

27 David Harel. Dynamic logic. In *Handbook of philosophical logic*, pages 497–604. Springer, 1984.

28 David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. doi:10.1016/0167-6423(87)90035-9.

29 Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. A Tutorial on Verifying LinkedList Using KeY. In *Deductive Software Verification: Future Perspectives – Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 221–245. Springer, 2020. doi:10.1007/978-3-030-64354-6_9.

30 Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.

31 J.P. Hollander. Verification of a model checking algorithm in VerCors, August 2021. URL: http://essay.utwente.nl/88268/.

32 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

33 Marieke Huisman and Raúl E. Monti. On the Industrial Application of Critical Software Verification with VerCors. In *Proceedings of Leveraging Applications of Formal Methods*, volume 12478 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2020. doi:10.1007/978-3-030-61467-6_18.

34 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

35 Reiner Hähnle. Private communication, July 2022.

36 Bart Jacobs. Modular Verification of Liveness Properties of the I/O Behavior of Imperative Programs. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, Proceedings*, volume 12476 of *Lecture Notes in Computer Science*, pages 509–524. Springer, 2020. doi:10.1007/978-3-030-61362-4_29.

37 Bart Jacobs. Private communication, March 2022.

38 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods – Third International Symposium, Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.

39 Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems – 8th Asian Symposium, Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010. doi:10.1007/978-3-642-17164-2_21.

40 Ioannis T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006. doi:10.1007/11813040_19.

41 Taekgoo Kim, Kevin Bierhoff, Jonathan Aldrich, and Sungwon Kang. Typestate protocol specification in JML. In *Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems*, pages 11–18. ACM, 2009. doi:10.1145/1596486.1596490.

42 Sophie Lathouwers and Marieke Huisman. Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers. In *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022*, pages 69–79. ACM, 2022. doi:10.1145/3524482.3527652.

43 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. doi:10.1145/1127878.1127884.

44 K Rustan M Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, 2009. doi:10.1007/978-3-642-00590-9_27.

45 K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009. doi:10.1007/978-3-642-03829-7_7.

46 Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. doi:10.1109/2.161279.

47 João Mota, Marco Giunti, and António Ravara. Java Typestate Checker. In *Proc. of Coordination Models and Languages (COORDINATION)*, volume 12717 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2021. doi:10.1007/978-3-030-78142-2_8.

48 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.

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

50 Peter O'Hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.

51 Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.

52 Wytse Hendrikus Marinus Oortwijn. *Deductive techniques for model-based concurrency verification*. PhD thesis, University of Twente, 2019.

53 Matthew J. Parkinson and Alexander J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In *Proceedings of Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 439–458. Springer, 2011. doi:10.1007/978-3-642-19718-5_23.

54 Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems, Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 158–182. Springer, 2015. doi:10.1007/978-3-662-46669-8_7.

55 John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. doi:10.1109/lics.2002.1029817.

56 R.B. Rubbens. Improving Support for Java Exceptions and Inheritance in VerCors. Master's thesis, University of Twente, 2020. URL: http://essay.utwente.nl/81338/.

57 Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, 159, 2020. doi:10.1016/j.jss.2019.110450.

58 Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In *Fundamental Approaches to Software Engineering, 11th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008. doi:10.1007/978-3-540-78743-3_19.

59 Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.

60 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.

61 André Trindade, João Mota, and António Ravara. Typestate Editor. https://typestate-editor.github.io/, 2022.

62 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.