# Java Typestate Checker

João Mota[1,2][0000−0003−3182−2245], Marco Giunti[1,3][0000−0002−7582−0308], and António Ravara[1,4][0000−0001−8074−0380]

[1] NOVA LINCS and NOVA School of Science and Technology, Portugal
[2] jd.mota@campus.fct.unl.pt
[3] marco.giunti@gmail.com
[4] aravara@fct.unl.pt

**Abstract.** Detecting programming errors and vulnerabilities in software is increasingly important, and building tools that help developers with this task is a crucial area of investigation on which the industry depends. In object-oriented languages, one naturally defines stateful objects where the safe use of methods depends on their internal state; the correct use of objects according to their protocols is then enforced at compile-time by an analysis based on behavioral types.
We present Java Typestate Checker (JATYC), a tool based on the Checker Framework that verifies Java programs with respect to typestates. These define the object's states, the methods that can be called in each state, and the states resulting from the calls. The tool offers the following strong guarantees: sequences of method calls obey to object's protocols; completion of objects' protocols; detection of null-pointer exceptions; and control of the sharing of resources through access permissions.
To the best of our knowledge, there are no research or industrial tools that offer all these features. In particular, the implementation of sharing control in a typestate-based tool seems to be novel, and has an important impact on programming flexibility, since, for most programs, the linear discipline imposed by behavioral types is too strict.
Sharing of objects is enabled by means of an assertion language incorporating fractional permissions; to lift from programmers the burden of writing the assertions, JATYC infers all of these by building a constraint system and solving it with Z3, producing general assertions sufficient to accept the code, if these exist.

**Keywords:** Behavioral types · object-oriented programming · typestates · access permissions · inference

## 1 Introduction

Programming errors such as de-referencing null pointers, or using resources wrongly, e.g., reading from a closed file, just to name a few, result in programs that might malfunction in many ways, producing unexpected behaviors or even crashing. It is, therefore, crucial to develop tools that assist the software development process by detecting mistakes as early as possible since these bugs occur more often than one might think [31].

In programming languages, some common errors are detected thanks to type systems implemented in type checkers [8]. Unfortunately, the subset of errors detected in present mainstream languages is still limited. For instance, most *OOP* languages, including Java, do not statically ensure that methods are called according to a specified protocol, like calling *hasNext* before calling *next* in an iterator. Usually, the protocol is specified in natural language in the documentation, but not statically enforced: this is a source of many errors, like accessing a variable that was not initialized [4]. More subtle undetected errors include concurrent threads reading and eventually closing a shared resource unexpectedly. While some language frameworks support a refined analysis, they require expert users to provide complex specifications, for example, in separation logic [21,29,20].

In this paper, we provide a tool to help filling this gap and introduce *Java Typestate Checker (JATYC)*, which type-checks a Java program where objects are associated with typestates. Java classes are annotated with typestates defining the behavior of class instances in terms of available methods and state transitions. With *JATYC*, well-typed programs have the following properties: objects are used according to their protocols (typestates); protocols reach the *end* state; null-pointer exceptions are not raised; data-races at the level of variables/fields and interference between method calls on the same object do not occur. Ensuring these properties is crucial to avoid protocol bugs like one found in [33], where an app tracing COVID-19 failed to perform a crucial step in the protocol: notify users if they were in close contact with potentially infectious patients, leaving the protocol uncompleted.

*JATYC* is a new implementation of Mungo [24] that adds critical features and fixes known issues, like assuming that a *continue* statement jumps to the beginning of the loop's body, thus skipping the condition expression [25], which may produce false negatives. *JATYC* was implemented in Kotlin [22] as a plugin for the Checker Framework [28].

Originally, Mungo was implemented with JastAdd [13], an extension to Java that supports a specification formalism called Rewritable Circular Reference Attributed Grammars [12], enabling the modular implementation of compiler tools and languages [13]. Unfortunately, JastAdd does not seem to be actively maintained[5] and editor support is lacking (except for syntax highlighting)[6]. The Checker Framework [28] is a tool that supports adding type systems to the Java language. With a plugin written in Java or any other Java interoperable language, one defines the type qualifiers and enforces the semantics of the type system. Programmers can then write the type qualifiers in their programs, with Java annotations[7], and use the plugin to detect errors [28]. The Checker Framework is actively maintained, well-integrated with the Java language and toolset, and has being used to detect bugs in popular projects, like null-pointer errors

[5] The second to last release was on 2019: `https://jastadd.cs.lth.se/releases/jastadd2/2.3.4/release-notes.php`

[6] `https://jastadd.cs.lth.se/web/tool-support/`

[7] `https://www.oracle.com/technical-resources/articles/java/ma14-architect-annotations.html`

in *Google Collections* [11], with their *Nullness Checker* plugin[8]. Together with Kotlin, it allowed us to be more productive in the development of the tool.

*JATYC* was developed in the context of the first author's master's dissertation [25]. The document and the code are freely distributed[9]. The major contributions with respect to the current version of Mungo are:

- checking the **absence of null pointer errors**, which is critical to avoid the "The Billion Dollar Mistake" [18];
- checking that the **protocols of objects are completed**, i.e. protocols reach the *end* state;
- support for the **static control of sharing** of objects, allowing safe aliasing and concurrency, while preventing data-races, patterns which are very common in object-oriented programming languages, thus increasing expressiveness and programming flexibility.

## 2 Related work

We are interested in object-oriented languages where well-typed programs follow these properties: objects follow their specified protocols; protocols reach the *end* state; null-pointer exceptions are not raised; and data-races do not occur. We now present a review of relevant works we know on the topics. A more complete overview of existing works is also available in the first author's master's dissertation [25].

**Behavioral Types** are type disciplines that describe properties associated with the behavior of programs [19]. Type systems that include this notion, allow for the static verification of interactions and protocol compliance, like ensuring that the *hasNext* method is called before *next* in an iterator object.

If multiple references to the same object exist (i.e. aliasing), type information can get outdated if the object changes state via another reference. In solutions that implement **behavioral types**, it is common to force the **linear use of objects**, meaning that there is only one reference for each object [2]. Unfortunately, this restricts what a programmer can do, since sharing references is common practice in imperative and object-oriented programming languages.

One solution to statically verify code with shared data is the use of **access permissions** [6,5]: abstract capabilities that characterize the way a shared resource can be accessed by multiple references [30]. This notion is built on Linear Logic [16], which treats permissions as linear resources, and Separation Logic [29,27], which reasons about program behavior against specifications. Access permissions are used to ensure that only a reference can write on a particular location at any given time, and to ensure that if a location is read by a thread, all other threads only have read permission for that location, thus avoiding interference in concurrent programs [30].

**Fractional permissions** [6] are concrete fractional numbers, ranged over 0 and 1, representing the permission for a shared resource: absence of permission

[8] `https://checkerframework.org/manual/#nullness-checker`
[9] `https://github.com/jdmota/java-typestate-checker`

is represented by 0; full permission (to read and write) is represented by 1; and shared read-only access is represented by a value strictly between 0 and 1. Fractional permissions can be split into a number of fractions and distributed among multiple references. For example, a permission $s$ can be split into $s_1$ and $s_2$ such that $s = s_1 + s_2$, allowing two references to have read access to the same resource. Permissions that were split may also be joined again [30,6].

**Mungo** [24] is a tool that extends Java with typestate definitions [15] which are associated with Java classes and define the behavior of instances of those classes, specifying the sequences of method calls allowed in terms of a state machine. Mungo then statically checks that method calls happen in order, following the specified behavior, and ensures that protocols reach the *end* state. Mungo does not allow aliasing of objects associated with typestates [24].

**Fugue** [10] integrates typestates [15] with an object-oriented programming language, allowing the programmer to add declarative specifications on interfaces, providing preconditions and postconditions, and marking methods that are used for allocating or releasing resources, thus limiting the order in which object's methods are called. Fugue then ensures that methods are called in correct order, preconditions are met before a method is called, and resources are not used before allocated or after being released. Fugue allows aliasing through its guarded types (*NotAliased* and *MayBeAliased*), which track the lifetime but not the number of references to an object [10].

**Plaid** [32,26,17] is a typestate-oriented programming language [15] designed for concurrency. In Plaid, the class of an object represents its current state, and that class can change dynamically during runtime. Not only the interface (i.e. available methods) depends on the state, the behavior (i.e. implementation) also depends on the current state. Plaid also incorporates access permissions, which are associated with each type to express the aliasing and the mutability of the corresponding object, using keywords such as *unique*, *shared* and *immutable*. Unfortunately, Plaid does not seem to be maintained any longer [30,17]. As far as we know, Plaid as no notion of protocol completion.

To statically ensure the absence of **null-pointer exceptions**, there are tools such as the *Nullness Checker* of the Checker Framework. This tool enhances the Java's type system so that types are non-nullable by default, which means that *null* values cannot be assigned to them. To declare a variable or field with a nullable type (i.e. a variable or field where the *null* value can be assigned to), one can use the *Nullable* annotation[10]. Some modern languages, such as Kotlin, also distinguish non-null types from nullable types, thus avoiding these exceptions[11]. Nonetheless, these may produce false positives that might force the programmer to provide additional checks, following a style known as *defensive programming*, that a value is not *null*, even when it is provable that the code is safe.[12]

[10] `https://checkerframework.org/manual/#nullness-checker`
[11] `https://kotlinlang.org/docs/null-safety.html`
[12] An example is available at `https://tinyurl.com/2hmwx7vk`.

## 3 Motivating example

To motivate the need for *JATYC*, consider a *LineReader* Java class that is responsible for both opening a file and reading it line by line.[13] List. 1.1 presents an implementation.

**Listing 1.1.** *LineReader* class

```
1  import java.io.*;
2  public class LineReader {
3    private FileReader file = null;
4    private int curr;
5
6    public Status open(String f) {
7      try {
8        file = new FileReader(f);
9        curr = file.read();
10       return Status.OK;
11     } catch (IOException exp) {
12       return Status.ERROR;
13     }
14   }
15
16   public String read()
17     throws IOException {
18     StringBuilder str =
19       new StringBuilder();
20     while (
21       curr != 10 && curr != -1
22     ) {
23       str.append((char) curr);
24       curr = file.read();
25     }
26     if (curr == 10)
27       curr = file.read();
28     return str.toString();
29   }
30
31   public boolean eof() {
32     return curr == -1;
33   }
34
35   public void close()
36     throws IOException {
37     file.close();
38   }
39
40   public enum Status { OK, ERROR }
41 }
```

The intended protocol is defined implicitly by the sequences of method calls that are supported, and by the "states" reached via those calls. To use the *LineReader*, one must invoke the *open* method passing the path of the file. If the call returns *ERROR*, then the file could not be opened. If it returns *OK*, then one can proceed to read the file. Before calling the *read* method, one must call the *eof* method to ensure that the end of the file was not reached. Each *read* call returns a string with a new line. After reading the file, the *close* method must be called to free the resources and close the underlying stream.

If this contract is not followed, errors may occur or wrong results may be produced. If one attempts to *read* before calling *open*, a *NullPointException* will

[13] The class could be a subclass of the abstract class java.io.Reader.

be thrown since the *file* field has a null reference (line 24). Additionally, if one calls the *read* method after calling *close*, an *IOException* will occur since the stream is closed (line 37). Finally, if one keeps reading the file after *eof* returns *true*, then *read* will return empty strings, giving a false impression that the file being read contains empty lines.[14] While the Java compiler accepts most of the wrong behaviors described above, in the next section we will show how to enrich Java programs with *typestate annotations* that allow rejecting programs containing these kinds of behavioral errors at compile-time.

## 4 What is the tool good for?

**Protocols.** All instances of a Java class having a typestate are checked in order to enforce the prescribed behavior. The typestate specifications are written in *.protocol* files, with the form *typestate* $T\{S_1 \dots S_n\}$, where each state $S_i$ is a list of *method transitions* $\{M_1 \dots M_j\}$, and the general form of $M_i$ is: `T m(T1,...,Tk) : <v1: S1, ..., vm: Sm>`, with $T, T_1, \dots, T_k$ Java types: when *method* $m$ is executed and returns value $v_i$, the typestate switches to state $S_i$.[15]

List. 1.2 presents the protocol for the *LineReader* (cf. List. 1.1). It specifies four states, *Init*, *Open*, *Read* and *Close*, and implicitly includes the *end* state, which is the final state. In the initial state *Init*, only the *open* method is available to be called (line 3). If the method returns *OK*, the state changes to *Open*; otherwise, the state changes to *end*, where no operations are allowed. After opening the file, the *close* method may be called anytime, except if the file was already closed (lines 7, 11, and 14). In the *Open* state, one may call the *eof* method (line 6). If it returns *true*, the state changes to *Close*; otherwise, the state changes to *Read*. In the *Read* state one may call the *read* method, which then changes the state to *Open* (line 10).

**Listing 1.2.** *LineReader* protocol

```
1  typestate LineReaderProtocol {
2    Init = {
3      Status open(String): <OK: Open, ERROR: end>
4    }
5    Open = {
6      boolean eof(): <true: Close, false: Read>,
7      void close(): end
8    }
9    Read = {
10     String read(): Open,
11     void close(): end
12   }
13   Close = {
14     void close(): end
15   }
16 }
```

To associate a protocol with a Java class, one must include a *Typestate* annotation containing the (relative) path of the protocol file. For backwards-

[14] Code examples are available online at `https://git.io/JtR7E`.
[15] The complete grammar is available at `https://git.io/JtMu3`.

compatibility with Mungo, we support the *Typestate* annotation from the *mungo.lib* package (List. 1.3).

**Listing 1.3.** *LineReader* class with *Typestate* annotation

```
1 import mungo.lib.Typestate;
2 @Typestate("LineReader.protocol")
3 public class LineReader { /* ... */ }
```

**Protocol compliance and completion.** *JATYC* ensures that instances of Java classes associated with a typestate not only obey to the corresponding protocol, but are also consumed (that is, they reach the *end* state): as a consequence, potentially important method calls are not forgotten and resources are freed. To add more flexibility, it is also possible to declare states in which an object may stop to be used. These *droppable states* [25] are declared by including the following special transition *drop: end*.[16]

To see an example of incorrect use of *LineReader*, consider List. 1.4, where errors are indicated in the comments. According to the protocol (List. 1.2), the reader object is in the *Close* state (line 5); thus, the only available method is *close*, that is, *read* is not available. Negating the loop condition fixes the error. Moreover, the *close* method is called nowhere: therefore the protocol does not reach the *end* state.

**Listing 1.4.** *LineReader* use

```
1  LineReader reader = new LineReader();
2  switch (reader.open()) {
3    case OK:
4      while (reader.eof()) {
5        System.out.println(reader.read());
6        // Error: cannot call "read" on state Close
7      }
8      break;
9    case ERROR:
10     System.err.println("Could not open file");
11     break;
12 }
13 // Error: object did not complete its protocol
```

**Nullness checking.** Null pointer errors are the cause of most runtime exceptions in Java programs [4,31]: being able to detect these errors at compile-time is therefore crucial. Towards that direction, *JATYC* offers the following guarantees: (1) types are non-null by default (contrary to Java's default type system[17]), method calls and field accesses are only performed on non-null types; (2) false positives (in classes associated to protocols) are ruled out by taking into account that methods are only called in a specific order. To allow a type to be nullable, one can use the *Nullable* annotation. The analysis is based on the formal work done in [7] for a language that served as basis for Mungo.

To exemplify guarantee (1), List. 1.5 presents two scenarios where methods are potentially called on null values. In line 5, *JATYC* reports an error since a method call could be performed on null. In line 9, no error is reported since the code checks for null first (line 8).

[16] Example of *droppable states* at `https://git.io/JOqfc`.

[17] The fact that null is a value of any type is the source of Java not being type safe. [1]

**Listing 1.5.** Nullness checking example (1)

```
1  import org.checkerframework.checker.jtc.lib.Nullable;
2  import java.io.FileReader;
3  public class Main {
4    void use1(@Nullable FileReader file) {
5      int c = file.read(); // Error: cannot call "read" on null
6    }
7    void use2(@Nullable FileReader file) {
8      if (file != null)
9        int c = file.read(); // Safe operation
10   }
11 }
```

To see how guarantee (2) works, consider List. 1.6: if one calls the *read* method before *open*, a null pointer error will occur. But since *open* must be called first (according to the protocol), we know that the *file* field is non-null when *read* is called; thus, the operation is safe. Notice the absence of *defensive programming*, required by many static analysis tools, namely by the *Nullness Checker* of the Checker Framework.

**Listing 1.6.** Nullness checking example (2)

```
1  import org.checkerframework.checker.jtc.lib.Nullable;
2  // ...
3  public class LineReader {
4    private @Nullable FileReader file = null;
5    // ...
6    public String read() {
7      // ...
8      curr = file.read(); // Safe operation
9      // ...
10   }
11 }
```

**Sharing.** In imperative languages, it is common to have multiple references to the same object. Aliasing makes it more difficult to track the state of each object, since it may change via another reference. The whole challenge becomes even harder in the presence of concurrent computations and accesses. Consider the example in List. 1.7 where a reference is stored in a field (line 1) and passed to a method call (line 5). Depending on the body of the *use* method, the program in List. 1.7 can be safe or unsafe: if the method modifies the state of the reader object (by calling methods on it), then the assumptions the *wrapper* made about the state of the stored reference are wrong, and in turn the program must be rejected, otherwise the program could be accepted. *JATYC* is able to track the potential state changes in the *use* method, thus allowing for a more liberal and sound management of resources.

**Listing 1.7.** Aliasing example (1)

```
1  class Wrapper { public LineReader reader = new LineReader(); }
2  class Main {
3    void main() {
4      Wrapper wrapper = new Wrapper();
5      use(wrapper.reader);
6    }
7    // ...
8  }
```

Assertion := Term | Term “∧” Assertion
Term := Access | Equality | TypeOf | Packed | Unpacked

Access := “access” “(” AccessLocation “,” f “)”
Equality := “eq” “(” Location “,” Location “)”
TypeOf := “typeof” “(” Location “,” t “)”
Packed := “packed” “(” Location “)”
Unpacked := “unpacked” “(” Location “)”

Location := id | id “.” Location
AccessLocation := id | id “.” “0” | id “.” AccessLocation

**Table 1.** Assertions' grammar

To track aliasing and control the operations that can be allowed, we integrate **behavioral types** [19] with **fractional permissions** [6] in an **assertion language**, obtaining an original (and promising) combination. Table 1 shows the grammar of assertions. Each assertion is a conjunction of five types of predicates: *access* (specifies the fractional permissions for *access locations*); *typeof* (asserts the current state of an object); *packed* (asserts that the object's fields are hidden behind the abstract typestate view); *unpacked* (asserts that the object's fields are exposed); *eq* (asserts that two locations point to the same object). *Access locations* refer to variables, fields, or the objects pointed by those, for example: $x$ refers to the local variable $x$; $x.y$ refers to the field $y$ of the object pointed by $x$; *x.0* refers to the object pointed by variable *x*. The *x.0* notion allows us to distinguish the permissions to call methods on objects from the permissions to read from or write to the variables or fields themselves.

**Listing 1.8.** Aliasing example (2)

```
1  LineReader r1 = new LineReader(), r2 = r1;
2  // access(r1, 1) ∧ access(r1.0, 1/2) ∧ typeof(r1, State “Init”) ∧
3  // access(r2, 1) ∧ access(r2.0, 1/2) ∧ typeof(r2, State “Init”) ∧
4  // packed(r1) ∧ packed(r2) ∧ eq(r1, r2)
```

List. 1.8 shows an assertion example for two variables with the same reference. The predicate *access(r1, 1)* indicates there is read and write access permission to the variable; *access(r1.0, 1/2)* indicates that there is only read permission to the object pointed by the variable (thus, only methods that keep the object in the same state may be called); *typeof(r1, State “Init”)* asserts that the object is in the *Init* state; and *packed(r1)* indicates that the object's fields are hidden behind the abstract typestate view. The same meanings apply to *r2*. Finally, *eq(r1, r2)* asserts that both variables hold the same reference.

To relieve the programmer from writing the assertions, *JATYC* embeds a prototypal **algorithm that infers all the assertions**. The **inference algorithm** is inspired by the work done in [14]. It has four steps: variables and fields are collected; assertions over symbolic fractions, types and equalities are constructed and associated with each expression in the code (before and after); each expression is analyzed and constraints over the symbols are produced; and

finally, the constraint system is given to the Z3 Solver [9]. A satisfiable system ensures: objects obey each other's protocols, even in the presence of aliasing; no data-races occur at the level of variables and fields; method calls that change the state of an object do not interfere with each other.

List. 1.9 shows a file being read in a separate thread (line 5), while in the main thread, the reader is closed before waiting for the thread to finish (line 9). This will result in an *IOException* when trying to read the closed file (line 5). The algorithm infers that full permission to the reader object is required in the thread (to read from it), and in the main thread (to close it). When *t.start* is called, full permission is acquired, leaving the main thread with no permission to the reader. When *r.close* is called, full permission is necessary, but not available. This contradiction will result in no solution being found, showing that there is a problem in the code. If *r.close()* is moved after *t.join()*, a solution will be found, and the code accepted, since *t.join()* gives back the permissions acquired when the thread started.

**Listing 1.9.** Concurrent *LineReader* use

```
1  LineReader r = new LineReader();
2  if (r.open() == Status.OK) {
3    Thread t = new Thread(() -> {
4      while (!r.eof()) {
5        println(r.read());
6      }
7    });
8    t.start();
9    r.close();
10   t.join();
11 }
```

**Limitations.** First, subtyping and dynamic method dispatch in Java are currently ignored. This means that the programmer currently needs to avoid using these Java features to still benefit from the guarantees that the tool provides. Secondly, the inference algorithm has some issues: all objects are considered to be unpacked, what causes problems if for example we want to work with recursive data structures; the analysis of threads only works if the thread is started and waited upon in the context in which it was created; and if the algorithm reports that it found no solution, no further information is given about the possible root problem. Thirdly, although the tool is fast at inferring the fractional permissions, it is slow at inferring the types. Finally, only concurrent scenarios with either a single reader and writer or multiple readers are allowed. For example, the scenario in List. 1.10 is not currently possible.

**Listing 1.10.** One writer and one reader

```
1  new Thread(() -> {
2    while (!r.eof())
3      println(r.read());
4  });
5
6  new Thread(() -> {
7    while (!r.eof()) {}
8  });
```

## 5 Future work

To lift the restrictions leading to the rejection of the code in List. 1.10, allowing more permissible yet safe concurrent accesses to data, we will incorporate standard approaches like Rely-Guarantee [23], locks and monitors. We also plan to fix limitations previously mentioned by taking into account subtyping and dynamic method dispatching. We are aware of the work done in [3], and it would be interesting to integrate the notion of synchronous session subtyping in *JATYC*. Subtyping support is crucial and it should be our first next step. Additionally, we plan to implement the inference of packing and unpacking, and pinpoint the code locations that caused the constraint system to be unsatisfiable, allowing the programmer to find out where the problem is. Furthermore, we would like to support generics and collections. Finally, we plan to improve the performance of the inference algorithm by only using Z3 to infer the fractional permissions and using a different technique to infer the types.

## Acknowledgements

We warmly acknowledge the anonymous reviewers whose comments and suggestions pushed us to present a more complete and well-rounded discussion of the topics.

This work was partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI) and by NOVA LINCS (UIDB/04516/2020) via the Portuguese Fundação para a Ciência e a Tecnologia.

## References


1. Amin, N., Tate, R.: Java and Scala's type systems are unsound: The existential crisis of null pointers. ACM Sigplan Notices **51**(10), 838–848 (2016). https://doi.org/10.1145/3022671.2984004
2. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P.M., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., et al.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3), 95–230 (2016). https://doi.org/10.1561/2500000031
3. Bacchiani, L., Bravetti, M., Lange, J., Zavattaro, G.: A session subtyping tool (2021), to appear in: 23rd International Conference on Coordination Models and Languages.
4. Beckman, N.E., Kim, D., Aldrich, J.: An empirical study of object protocols in the wild. In: European Conference on Object-Oriented Programming. pp. 2–26. Springer (2011). https://doi.org/10.1007/978-3-642-22655-7_2
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: The 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 259–270 (2005). https://doi.org/10.1145/1040305.1040327
6. Boyland, J.: Checking interference with fractional permissions. In: International Static Analysis Symposium. pp. 55–72. Springer (2003). https://doi.org/10.1007/3-540-44898-5_4
7. Bravetti, M., Francalanza, A., Golovanov, I., Hüttel, H., Jakobsen, M.S., Kettunen, M.K., Ravara, A.: Behavioural types for memory and method safety in a core object-oriented language. In: Asian Symposium on Programming Languages and Systems. pp. 105–124. Springer (2020)
8. Cardelli, L.: Type systems. ACM Computing Surveys **28**(1), 263–264 (1996). https://doi.org/10.1145/234313.234418
9. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
10. DeLine, R., Fähndrich, M.: The fugue protocol checker: Is your software baroque. Tech. rep., Technical Report MSR-TR-2004-07, Microsoft Research (2004)
11. Dietl, W., Dietzel, S., Ernst, M.D., Muşlu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: The 33rd International Conference on Software Engineering. pp. 681–690 (2011). https://doi.org/10.1145/1985793.1985889
12. Ekman, T., Hedin, G.: Rewritable reference attributed grammars. In: European Conference on Object-Oriented Programming. pp. 147–171. Springer (2004). https://doi.org/10.1007/978-3-540-24851-4_7
13. Ekman, T., Hedin, G.: The jastadd extensible java compiler. In: The 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. pp. 1–18 (2007). https://doi.org/10.1145/1297105.1297029
14. Ferrara, P., Müller, P.: Automatic inference of access permissions. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 202–218. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_14
15. Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. ACM Transactions on Programming Languages and Systems (TOPLAS) **36**(4), 12 (2014). https://doi.org/10.1145/2629609
16. Girard, J.Y.: Linear logic. Theoretical computer science **50**(1), 1–101 (1987). https://doi.org/10.1016/0304-3975(87)90045-4

17. Group, T.P.: The plaid programming language - introduction, https://www.cs.cmu.edu/~aldrich/plaid/plaid-intro.pdf, accessed: 2021-04-10
18. Hoare, T.: Null references: The billion dollar mistake (2009), https://tinyurl.com/eyipowm4, presentation at QCon London
19. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P.M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., et al.: Foundations of session types and behavioural contracts. ACM Computing Surveys (CSUR) **49**(1), 1–36 (2016). https://doi.org/10.1145/2873052
20. Ishtiaq, S.S., O'hearn, P.W.: Bi as an assertion language for mutable data structures. In: The 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 14–26 (2001). https://doi.org/10.1145/1988042.1988050
21. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods Symposium. pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
22. Jemerov, D., Isakova, S.: Kotlin in action. Manning Publications Company (2017)
23. Jones, C.B.: Specification and design of (parallel) programs. In: Mason, R.E.A. (ed.) Information Processing 83, The IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983. pp. 321–332. North-Holland/IFIP (1983)
24. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with mungo and stmungo. In: The 18th International Symposium on Principles and Practice of Declarative Programming. pp. 146–159. ACM (2016). https://doi.org/10.1145/2967973.2968595
25. Mota, J.: Coping with the reality: adding crucial features to a typestate-oriented language. Master's thesis, NOVA School of Science and Technology (2021), https://github.com/jdmota/java-typestate-checker/blob/master/docs/msc-thesis.pdf
26. Naden, K., Bocchino, R., Aldrich, J., Bierhoff, K.: A type system for borrowing permissions. ACM SIGPLAN Notices **47**(1), 557–570 (2012). https://doi.org/10.1145/2103621.2103722
27. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: International Workshop on Computer Science Logic. pp. 1–19. Springer (2001). https://doi.org/10.1007/3-540-44802-0_1
28. Papi, M.M., Ali, M., Correa Jr, T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for java. In: The 2008 international symposium on Software testing and analysis. pp. 201–212 (2008). https://doi.org/10.1145/1390630.1390656
29. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. IEEE (2002). https://doi.org/10.1109/lics.2002.1029817
30. Sadiq, A., Li, Y.F., Ling, S.: A survey on the use of access permission-based specifications for program verification. Journal of Systems and Software **159**, 110450 (2020). https://doi.org/10.1016/j.jss.2019.110450
31. Sunshine, J.: Protocol programmability. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2013)
32. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, É.: First-class state change in plaid. ACM SIGPLAN Notices **46**(10), 713–732 (2011). https://doi.org/10.1145/2076021.2048122
33. Wetsman, N.: Contact tracing app for england and wales failed to flag people exposed to covid-19. The Verge (2020), https://www.theverge.com/2020/11/2/21546618/uk-coronavirus-contact-tracing-app-error-alert-isolation