The Journal of Systems and Software 211 (2024) 111972

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Survey of annotation generators for deductive verifiers ☆

Sophie Lathouwers *, Marieke Huisman

University of Twente, EEMCS-FMT (Zilverling), P.O. Box 217, 7500AE Enschede, The Netherlands

ARTICLE INFO

Dataset link: https://doi.org/10.4121/9c83933e-8406-4e49-ac4d-1f8bb55ed988

Keywords:
Deductive verification
Specifications
Annotations
Specification inference
Specification generation
Survey

ABSTRACT

Deductive verifiers require intensive user interaction in the form of writing precise specifications, thereby limiting their use in practice. While many solutions have been proposed to generate specifications, their evaluations and comparisons to other tools are limited. As a result, it is unclear what the best approaches for specification inference are and how these impact the overall specification writing process. In this paper we take steps to address this problem by providing an overview of specification inference tools that can be used for deductive verification of Java programs. For each tool, we discuss its approach to specification inference and identify its advantages and disadvantages. Moreover, we identify the types of specifications that it infers and use this to estimate the impact of the tool on the overall specification writing process. Finally, we identify the ideal features of a specification generator and discuss important challenges for future research.

1. Introduction

Deductive verification is a technique that can help to develop reliable systems, even in the presence of features such as exceptions, concurrency and unbounded data. In deductive verification, the user needs to provide a program and a specification describing the intended behavior of the program. Given such a program and specification, a program logic is used to prove that the program adheres to the specification. However, as with all powerful techniques, this comes at a cost. For deductive verification, the user is required to provide specifications, typically expressed as annotations in the code.

The specifications can be separated into two groups: (1) the main specification describing the expected behavior of the program, and (2) auxiliary annotations which are needed by the computer to solve the proof, such as loop invariants to reason about loops. Ideally, the main specification of the program is independent of the actual implementation and could be reused for different implementations. Such specifications are typically written in the contracts of public methods. Auxiliary specifications can be closely related to the implementation as different implementations may require different proof steps. These specifications are typically found within methods themselves or as contracts of private methods. In this work, we focus on the generation of both the main and the auxiliary specifications.

Proving the correctness of a program with a deductive verifier, is typically an iterative process. The user starts by providing a program and its specification to the verifier. The verifier will then report whether the program adheres to the specification or not. If the program does not adhere to the specification, then the user needs to reflect to determine whether there is a bug in the program *or* in the specification. After repairing the bug, the user repeats the process to see whether the program can now be proven correct.

This verification process requires significant amounts of effort from the user. Especially finding the right specifications can be very difficult. Previous case studies have shown that the specification writing burden can become as big as 10:1 (10 lines of specification per 1 line of code) which hinders (Baumann et al., 2012; Beckert and Hähnle, 2014; Hähnle and Huisman, 2019) the adoption of deductive verification.

To alleviate the specification writing burden on the user, a commonly proposed solution (Filliâtre and Marché, 2007; Gurov et al., 2017; Huisman and Monti, 2020; Knüppel et al., 2018; Scheben, 2014) is to generate (part of) the required specifications. However, it is currently unclear what the best approach for inference is as tools use their own benchmarks for evaluation and do not measure their impact on the overall specification writing process. As a result, it is impossible to compare the results of the various tools.

To address this problem, we provide an overview of the state-of-the-art specification inference tools. We include tools that generate specifications for deductive verifiers, such as preconditions and post-conditions, and that support inference for Java programs. For each tool we describe the techniques that are used to generate specifications and the type of specifications that they can infer (Section 3). Next, we use a taxonomy and its analysis described in Lathouwers and Huisman (2022) to estimate the impact of each tool (Section 4) on the overall specification writing process. We first identify the type of specifications that the

☆ Editor: Dr. Martin Pinzger.

* Corresponding author.
E-mail addresses: sophie.lathouwers@gmail.com (S. Lathouwers), m.huisman@utwente.nl (M. Huisman).

https://doi.org/10.1016/j.jss.2024.111972
Received 14 July 2023; Received in revised form 5 December 2023; Accepted 19 January 2024
Available online 10 February 2024

tool infers, and then look up how often these types of specifications are expected to occur. This provides an estimate of the maximum impact of each tool on the complete specification writing process. Combining all these results allows us to identify which types of specifications can be inferred and where the gaps are in the current state of the art. We use this to outline features of an ideal specification inference tool in Section 5, thereby providing practical recommendations for the future development of inference tools. Related work, including some tools that were not included in this survey because of our selection criteria, is discussed in Section 6. Finally, we discuss open research problems in Section 7 and conclude in Section 8.

To summarize, the contributions of this paper are:

- An overview of specification inference tools for Java programs that infer specifications for deductive verifiers;
- An analysis of the impact of the specification inference tools on the overall specification writing process;
- Practical recommendations for the future development of inference tools by discussing what an ideal inference tool looks like; and
- A list of open research problems for specification inference.

2. Background

In this section we will briefly introduce different types of specifications that are commonly used in deductive verification. Many deductive verifiers are based on or use extensions of Hoare logic. In Hoare logic, a program is typically specified as a triple of the form: `{P}S{Q}`. In this triple, `P` is called the precondition, `S` is the program and `Q` is the postcondition. The idea is that, if condition `P` holds, and we execute the program `S`, then we can prove that `Q` holds after the execution of the program. This assumes that the program terminates, thus it only guarantees partial correctness. While there exist approaches that reason about complete correctness, i.e. including termination, nearly all specification inference tools focus solely on inferring functional properties, which are used in both approaches. Pre- and postconditions are typically described on a method-level meaning that if precondition `P` holds when calling the method, then we can prove that postcondition `Q` holds after executing the method. If the precondition does not hold when calling the method, then nothing is guaranteed at the end of the method.

If a postcondition expresses a property that should hold in the case of exceptional behavior, i.e. a method terminates by throwing an exception, this is called an exceptional postcondition.

A (class) invariant is a condition that is established during the construction of an object and is afterwards maintained between method calls.

A loop invariant is a specification that expresses a property for a specific loop. This property should hold when entering the loop, as well as after each loop iteration.

Assertions are similar to assertions in program code; they express that a property should hold at the location of the assertion.

For more extensive definitions of these concepts, we refer the interested reader to Leavens et al. (2008) and Meyer (2002). Leavens et al. (2008) describe the meaning of these concepts in the Java Modeling Language (JML) (Leavens et al., 2006), a language that is often used for the specification of Java programs. Meyer (Meyer, 2002) introduces the fundamentals of the Design-by-Contract (DbC) approach which includes the use of contracts to specify a program's intended behavior.

Specifications are typically used to describe the behavior of a program, e.g. to say whether the method is expected to terminate or what the value of a variable is. However, there are also other types of specifications, such as permission specifications (Lathouwers and Huisman, 2022) and frame conditions, which describe whether a method has access to a variable. In JML (Leavens et al., 2006), such permissions are described in `assignable` and `accessible` clauses at the method-level. `assignable` clauses state that the method may assign values to all locations named in the assignable clause. `accessible` clauses state that the method may read values from all locations named in the accessible clause. Several separation logic-based tools, e.g. VerCors (Blom et al., 2017) and Verifast (Jacobs et al., 2011), express the ability to access a variable through permissions which are predicates that can be used in any other specification location such as preconditions, postconditions and loop invariants. For example, the precondition `requires Perm(x, write)` expresses that the method can assign to variable `x` in VerCors' specification language.

3. Overview of specification generators

In this section, we will provide an overview of the tools that are available for the inference of annotations for deductive verification of Java programs. We have selected tools that infer annotations for Java programs, which allows us to use earlier work (Lathouwers and Huisman, 2022) to estimate their maximum impact. Moreover, we selected tools that generate annotations that can be used by deductive verifiers or can be used with little additional effort. Tools that do not fulfill these selection criteria, but that use interesting approaches or infer other types of specifications, will be discussed in Section 6.

The inference tools that we will discuss are:

- Daikon
- eChicory
- DynaMate
- Agitator
- EvoSpex
- SLING
- ALearner
- DIG/SymInfer
- Strongarm
- KeY
- "Mining preconditions of APIs in large-scale code corpus"
- Toradocu/Jdoctor
- C2S
- ChatGPT

We have divided these tools into three sections. Section 3.1 will focus on tools that execute the program for the inference process. Section 3.2 focuses on tools that rely on static inference techniques. And, Section 3.3 focuses on tools that rely on natural language processing.

For each tool we provide a brief explanation of the techniques that are used to infer specifications as well as the advantages and disadvantages of the approach. We were able to successfully run Daikon, EvoSpex, Strongarm, Toradocu and ChatGPT. For these tools, we also reflect on our experiences of using the tools on some examples.

We prepared three examples to test each tool's ability: a counter, a binary search and an arraylist. The code of these examples can be found in Listing 1 and Appendix A. The counter was chosen as it is a very simple example using pre- and postconditions. Binary search (see Listing 1) is slightly more complicated as it deals with arrays and quantified statements over the elements. Arraylist is even more complicated as its methods have some logical conditions and it uses exceptions. Together they cover typical applications, ranging from easy to logically complex.

We have verified each example beforehand with OpenJML which provides us with a baseline of the specifications that would be needed for verification (see Appendix A). The number of annotations used for verification per example are as follows: 32 for the counter, 16 for the binary search, and 103 for the arraylist. The total number of used annotations sums up to 151. Of these 151, 17 were required to avoid overflow and underflow warnings and four to avoid `PossiblyBadArrayAssignment` warnings by OpenJML. The other annotations include a wide variety of specifications, such as preconditions, (exceptional) postconditions, loop invariants, class invariants, assignable clauses, behavior clauses and decreases clauses.

To test the tools, each specification inference tool was provided the code without annotations. We have also prepared Javadoc documentation and a test suite as this was required for some tools. The Javadoc was written in the same style as the Javadoc for the List and ArrayList

```
1     public class BinarySearchGood {
2
3     //@ requires sortedArray != null;
4     //@ requires 0 < sortedArray.length < Integer.MAX_VALUE;
5     //@ requires \forall int i,j; 0 <= i < j < sortedArray.length; sortedArray[i] <=
      ↪ sortedArray[j];
6     //@ ensures 0 <= \result < sortedArray.length <==> (\exists int i; 0 <= i <
      ↪ sortedArray.length; sortedArray[i] == value);
7     //@ ensures \result != -1 ==> sortedArray[\result] == value;
8     //@ ensures \result == -1 <==> (\forall int i; 0 <= i < sortedArray.length; sortedArray[i] !=
      ↪ value);
9     //@ pure
10    public static int search(int[] sortedArray, int value) {
11        if (value < sortedArray[0]) return -1;
12        if (value > sortedArray[sortedArray.length-1]) return -1;
13        int lo = 0;
14        int hi = sortedArray.length-1;
15        //@ loop_invariant 0 <= lo < sortedArray.length;
16        //@ loop_invariant 0 <= hi < sortedArray.length;
17        //@ loop_invariant (\exists int i; 0 <= i < sortedArray.length; \old(sortedArray[i]) ==
      ↪ value) ==> sortedArray[lo] <= value <= sortedArray[hi];
18        //@ loop_invariant \forall int i; 0 <= i < lo; sortedArray[i] < value;
19        //@ loop_invariant \forall int i; hi < i < sortedArray.length; value < sortedArray[i];
20        //@ loop_decreases hi - lo;
21        while (lo <= hi) {
22            int mid = lo + (hi-lo)/2;
23            if (sortedArray[mid] == value) {
24                return mid;
25            } else if (sortedArray[mid] < value) {
26                lo = mid+1;
27            } else {
28                hi = mid-1;
29            }
30        }
31        return -1;
32    }
33 }
```

Listing 1: Binary search program that has been verified using OpenJML. Source: https://www.openjml.org/examples/bubble-sort.html

of Java 8. The test suite setup is inspired by the Daikon examples, where there is one test class for each program. Each test class contains one method for each method in the program. The ArrayList program has some additional methods to trigger edge cases such as getting the element of an invalid index. If input is needed to call a method, e.g. the index to get an element from, then this is randomly generated. The test suite will randomly choose which method to call.

The tools (Daikon, EvoSpex, Strongarm, Toradocu, ChatGPT) were run using their default settings. Our experiences with these tools on the three examples are discussed in the "Our experience" section of Daikon (Section 3.1.1), EvoSpex (Section 3.1.5), Strongarm (Section 3.2.2), Toradocu (Section 3.3.1) and ChatGPT (Section 3.3.3). Moreover, we provide an overview of these results in Section 3.4. An artifact containing a Virtual Machine with Daikon, EvoSpex, Strongarm and Toradocu installed, as well as the examples used to test each tool, is freely available at:

https://doi.org/10.4121/9c83933e-8406-4e49-ac4d-1f8bb55ed988

An overview of *all* tools that are discussed in this section can also be found in Appendix B. This overview includes the input that the tool requires, a brief methodology description, the type of specifications that the tool infers and whether the tool is publicly available.

3.1. Dynamic techniques

In this section we discuss the tools that rely on running the program to infer specifications, also known as dynamic techniques. Some of these tools combine static and dynamic approaches to infer specifications. This section includes the tools: Daikon, eChicory, DynaMate, Agitator, EvoSpex, SLING, ALearner and DIG/SymInfer. We have tested Daikon and EvoSpex on the counter, binary search and arraylist examples.

3.1.1. Daikon

Technique. We start with arguably the most well-known specification generator namely *Daikon* (Ernst et al., 2007). Daikon uses a dynamic approach where it observes several program executions. During these program executions it observes variables at specific points in the program, by default the entry and exit points of a method. Depending on the values of these variables at these points, Daikon will then instantiate an invariant based on a built-in grammar. These generated invariants are guaranteed to hold for the observed executions.

The Daikon grammar includes properties such as:

- `x OP y` where `OP` can be replaced by `==`, `!=`, `>=`, `>`, `<=`, `<`
- `x[i] OP x[i+1]` where `OP` can be replaced by `==`, `!=`, `>=`, `>`, `<=`, `<`
- `x[0] OP y[0] && x[1] OP y[1] && ... && x[n] OP y[n]` where `OP` can be replaced by `==`, `!=`, `>=`, `>`, `<=`, `<`
- For each element in the sequence `x[]`, `x[i] OP y`, where `OP` can be replaced by `==`, `!=`, `>=`, `>`, `<=`, `<`
- `x % y == 0`
- `x == y ** 2`
- `x == Multiply(y,z)`
- `a != null ==> a.x == 0`
- `ax + by + c == 0`
- x is a member of y[]
- x is a substring of y

```
1  public class BinarySearchGood {
2      public static int search(int[] sortedArray, int value) {
3          if (value < sortedArray[0]) return -1;
4          if (value > sortedArray[sortedArray.length-1]) return -1;
5          int lo = 0;
6          int hi = sortedArray.length-1;
7          while (lo <= hi) {
8              dummy_call(lo, hi, sortedArray, value);            int mid = lo + (hi-lo)/2;
9              if (sortedArray[mid] == value) {
10                 return mid;
11             } else if (sortedArray[mid] < value) {
12                 lo = mid+1;
13             } else {
14                 hi = mid-1;
15             }
16         }
17         return -1;
18     }
19
20     public static void dummy_call(int lo, int hi, int[] sortedArray, int value) {}}
```

Listing 2: Binary search program that has been modified such that Daikon can infer loop invariants. The method `dummy_call` (line 21) and a call to this method at the beginning of the loop (line 8) have been added.

```
1      /*@ requires lo >= 0; */
2      /*@ requires hi >= 0; */
3      /*@ requires sortedArray != null; */
4      /*@ requires (\forall int i, j; (0 <= i && i <= sortedArray.length-1 && 0 <= j && j <=
       ↪ sortedArray.length-1) ==> ((i+1 == j) ==> (sortedArray[i] < sortedArray[j]))); */
5      /*@ requires value != 0; */
6      /*@ requires lo <= hi; */
7      /*@ requires lo <= sortedArray.length-1; */
8      /*@ requires hi <= sortedArray.length-1; */
9      /*@ requires sortedArray[lo] <= sortedArray[hi]; */
10     /*@ ensures (\forall int i, j; (0 <= i && i <= sortedArray.length-1 && 0 <= j && j <=
       ↪ sortedArray.length-1) ==> ((i+1 == j) ==> (sortedArray[i] < sortedArray[j]))); */
11     /*@ ensures \old(lo) <= sortedArray.length-1; */
12     /*@ ensures \old(hi) <= sortedArray.length-1; */
13     /*@ ensures sortedArray[\old(lo)] <= sortedArray[\old(hi)]; */
14     public static void dummy_call(int lo, int hi, int[] sortedArray, int value) {}
```

Listing 3: Specifications that were generated for the `dummy_call` method using Daikon. Equivalent pairs of pre- and postconditions are considered to be loop invariants.

Unlike some other techniques, Daikon does not require any annotations to start with. By default it generates invariants, preconditions, postconditions and assignable clauses. While it can technically infer specifications at any point in the program, assertions at other program points, including loop invariants, require the user to either modify the front end or to use a workaround. The workaround requires the user to define a function (that does not do anything) to which they pass all variables of interest. Then the user should add a call to this dummy procedure at the location of interest, e.g. the top of a loop. An example of this workaround is illustrated in Listing 2.

If you apply such a workaround, Daikon will generate pre- and postconditions for the dummy procedure (see Listing 3). Daikon's documentation mentions that it will generate (identical) pre- and postconditions. These identical pre- and postconditions can then be (manually) transformed into loop invariants. However, it is unclear from the documentation what to do with pre- or postconditions for which there is no identical counterpart. In this work, we will only interpret a generated specification as a loop invariant if there is a pair of equivalent pre- and postconditions.

Advantages and disadvantages. The advantages of Daikon are that (1) it is easy to install and use, (2) it takes only a short time to infer specifications (in our experience), and (3) it can automatically add the specifications to your program.

The disadvantages of Daikon are that (1) it can only derive invariants that are expressible in the built-in grammar, (2) the inferred invariants are dependent on the quality of the executed tests, and (3) it may infer incorrect specifications (i.e. actual behavior instead of intent).

Our experience. Using Daikon, we generated 158 specifications, 20 for counter, 11 for binary search and 127 for arraylist. The average runtime, measured over 5 runs, for counter, binary search and arraylist were 27, 9 and 11 s respectively. Of the 158 generated specifications, 29 of them could be found in the baseline specifications. This covers 19% of the baseline specifications. Of the generated specifications, 83 (53%) were wrong, and 47 (30%) were correct but unnecessary. For the generated loop invariants we only counted equivalent pre- and postcondition pairs and disregarded the other additional clauses that were generated. The generated specifications included preconditions, postconditions, invariants, loop invariants and modifies[1] clauses. It generated mostly specifications such as `this.list != null` and `\result == -1`. It also managed to generate several

[1] These are equivalent to `assignable` clauses.

annotations with quantifiers such as `(\forall int i, j; (0 <= i && i <= sortedArray.length-1 && 0 <= j && j <= sortedArray.length-1) ==> ((i+1 == j) ==> (sortedArray[i] < sortedArray[j])));`.

Moreover, it takes into account some built-in keywords of JML such as `\result`, `\old` and `\typeof`.

Sometimes the inferred specifications were incorrect or not as precise as one would like, either because of the limited test suite or because of the limits of the grammar. For example, for a `contains` method of an `ArrayList` it inferred that the size of the list would always be zero. It also inferred `\result >= 1 && \result <= sortedArray.length -1` instead of `sortedArray[\result] == value || \result == -1`, even though there is a test case for the contains method with an element that is not in the list.

The workaround for generating loop invariants can be a little confusing. The documentation of Daikon indicates that it should generate identical pre- and postconditions for the dummy procedure. However, not all generated pre- and postconditions were identical. For example, for the binary search example Daikon generated only one set of identical pre- and postconditions. Another three specification pairs were identical except for an `\old` keyword around one or more variables. Aside from these, it also produced additional preconditions that did not have any equivalent corresponding postcondition. Specifications from this last category were disregarded and not included as loop invariants in our examples.

Finally, we note that some specifications would probably be written differently if they were written by a user, which impacts readability of the specifications. Users often write `this.value == \old(this.value)+1` whereas Daikon generated `this.value - \old(this.value) - 1 == 0`. While these are equivalent, the Daikon-generated version may be harder to read for users.

3.1.2. eChicory

Technique. After the success of Daikon, several tools were developed that either depended on or extended it. The first of these tools that we discuss is *eChicory* (Alsaeed and Young, 2018), an extension of Daikon.

Daikon, specifically the Chicory instrumentation, assumes that the variable tree structure does not change after a method is invoked. As a result, the set of variables it tracks is static, and it can only track value changes of variables known at invocation. eChicory improves upon this by also regularly evaluating the variables structure. This allows eChicory to also derive specifications about dynamically established relations, e.g. the relation between a view and model in the model-view-controller pattern.

Advantages and disadvantages. The main advantage of eChicory in addition to Daikon is that it can derive specifications about dynamically established behaviors.

The main disadvantages are the same as those for Daikon namely (1) it can only derive specifications expressible in Daikon's built-in grammar, (2) the inferred invariants are dependent on the quality of the test suite, and (3) it may infer the actual behavior of the program as opposed to the intent.

Our experience. While eChicory is available, we were unfortunately unable to run it successfully on our examples because it could not find or load the main class.

3.1.3. DynaMate

Technique. *DynaMate* (Galeotti et al., 2014) combines three different techniques to provide fully automatic verification of programs. The user needs to provide the code and method contracts (pre- and postconditions). DynaMate will then first use a test generator (EvoSuite (Fraser and Arcuri, 2011)) to generate several executions. These executions are passed to two dynamic invariant detector techniques (Daikon, Gin-Dyn) which use them to find loop invariant candidates. Daikon uses, as mentioned before, a grammar to generate loop invariants. Gin-Dyn was designed as part of DynaMate. It is based on the idea that "loop invariants can often be seen as weakened forms [of] postconditions" (Galeotti et al., 2014). Gin-Dyn takes the provided postconditions and syntactically mutates them. The loop invariant candidates are tested to see whether they are invalidated by any test runs. If a candidate was not invalidated, then the loop invariant is added to the program. This program is then passed to the verifier ESC/Java2 (Chalin et al., 2006) to try and statically prove its correctness. If it could not be proven correct, the process described above is repeated. EvoSuite will then be used to generate new tests that falsify the unproven loop invariant candidates.

Advantages and disadvantages. The advantage of DynaMate is that it can take over part of the work that the user would typically do. Specifically, the iterative process to find the correct loop invariants. Moreover, the loop invariants that are found are known to be provably correct.

This approach works well because it is limited to loop invariants, which can often be derived from postconditions. A similar approach for other types of specifications, such as preconditions, would probably be very difficult.

A disadvantage of the approach is that it may discard specifications that describe the intended behavior if the program has bugs. This is caused by the selection procedure which discards invariants that are invalidated by test runs.

Our experience. While the source code and scripts are still available through a project website, we were unfortunately not able to get DynaMate working. The Makefile results in some errors and when running the commands manually, Daikon is unable to find any program point declarations and as a result no invariants are generated.

3.1.4. Agitator

Technique. *Agitator* (Boshernitsan et al., 2006) was originally developed to support test practices rather than verification. It uses test-input generation to explore the program's actual behavior. It combines this with a dynamic invariant detection algorithm, similar to Daikon's algorithm, to find observations. These observations represent relationships between values that hold under various input values. Some examples are `this.getName() != null` and `0.0 <= this.getPrice() <= 1000.0`. These observations are shown to the user and the user can then choose to turn an observation into an assertion. As such, it is still up to the user to determine whether the observations describe the intended behavior.

Advantages and disadvantages. Agitator was developed while taking the developer's workflow into account, and has therefore been nicely integrated into the Eclipse interface.

One of the unique things about Agitator is that it leaves it up to the user to determine whether the proposed observations should become assertions. This can be considered an advantage, as the chosen assertions will describe the intended behavior instead of the actual behavior. However, one can also consider this a disadvantage as it requires more interaction from the user. In any case, it proposes possible assertions which the user may either not have found or would have taken longer to find.

In the evaluation described in the paper (Boshernitsan et al., 2006), it is mentioned that 11% of the observations were useful invariants. It is unclear whether the proposed workflow, which includes generating test inputs, running tests and selecting assertions, is more time efficient than writing the specifications as is typically done.

Agitator generates assertions instead of annotations and thus will require some additional work to transform the assertions into contracts.

Our experience. There does not seem to be a publicly available artifact for Agitator.

3.1.5. EvoSpex

Technique. *EvoSpex* (Molina et al., 2021) aims to address a limitation of Daikon namely the grammar used to infer specifications. It specifically aims to generate more complex properties for reference-based implementations such as structural constraints and membership properties.

EvoSpex starts by generating tests (bounded exhaustively) for the given program. These tests are then executed during which it observes the program state. This provides valid (corresponding to actual behavior) pre and post states of the program. These states are then mutated to obtain invalid (not corresponding to observed behaviors) pre and post states. Given these valid and invalid pre/post states, it uses a genetic algorithm to generate a postcondition that satisfies the valid states and does not satisfy the invalid states.

Advantages and disadvantages. Starting with the positives, EvoSpex is less affected by specific values observed in executions compared to Daikon. Moreover, it is able to generate some more complex properties for reference-based implementations. Whether this is useful, depends on the program for which one wants to infer specifications.

The disadvantages are that (1) EvoSpex takes more time than Daikon to infer specifications, (2) it may infer actual behavior as opposed to intent, and (3) while it can generate more complex properties than Daikon in some cases, it is still limited by its genetic algorithm operators and may not generate e.g. complex arithmetic properties.

Our experience. When running EvoSpex on the examples, several errors occurred, including an stmfootnotesize IndexOutOfBoundsException, a FileNotFoundException and a ClassCastException, due to which we were unable to generate specifications for the constructor of `Counter` (`Counter()`) and three methods of the `ArrayList` example (`ArrayList()`, `enlarge()` and `add()`). In total, EvoSpex generated 25 annotations, 8 of which also occurred in our baseline specifications. These 8 postconditions covered 4% of all baseline specifications. If we compare it to the 30 postconditions in the baseline, it covered 20%. It generated 6 (24%) incorrect specifications, and 11 (44%) correct but unnecessary specifications.

All specifications generated by EvoSpex for our examples were postconditions and seemed relatively simple. It does support keywords similar to `\old` and `\result`.

EvoSpex generates less specifications as well as less complex specifications than Daikon for our examples. It seemed capable of generating most required postconditions for the counter which was our easiest example. However, for the binary search example it was only capable of generating a simple null check on the class itself. For the Arraylist example, the generated postconditions were all related to the size of the list or a null check for the `list` array. It did not include any postconditions about elements of the array.

The specifications can be verbose as a result of using a genetic algorithm. For example, take this specification that was generated for a method that decreased a counter by a given value: `ensures this_pre.value != this.value - this_pre.value - 1 + this.value + this.value + this_pre.value + 1 + 1`. This is equivalent to `ensures this_pre.value!=3*this.value+1` as some of the additions/negations cancel each other. These additions/subtractions can be traced back to the numeric addition/subtraction operator that is used in the genetic algorithm. If one wants to use EvoSpex in practice, it can therefore be useful to implement a rewriter which simplifies expressions.

There are some minor inconveniences in the current implementation that makes EvoSpex less nice to work with. It has a limited support for the language, for example, properties over arrays are not yet supported. As a result, EvoSpex is unable to generate a lot of the more interesting postconditions for the binary search and arraylist example. While the annotations are expressible in JML, they are not generated in this format so some manual translation is needed. And, it does not have the ability to automatically add the specifications to your program. We also ran into several exceptions while running the tool caused by a bug in one of the libraries that is used.

3.1.6. SLING

Technique. *SLING* (Le et al., 2019) is a tool that aims to find assertions for dynamically allocated data structures. Specifically, they target shape properties (over heap variables) and equality constraints (over stack variables). They do not generate disjunctive properties or numerical relations. Unlike most other tools, SLING infers properties in separation logic.

SLING expects four things as input: the program, a location of interest, a set of predefined predicate templates and a set of sample inputs (or test suite). Given these inputs, SLING first runs the program on the provided sample inputs. During these runs, it captures memory information at the location of interest in the trace. It iteratively analyzes these traces to compute properties that should hold at the location of interest. For each pointer variable, it generates a separation logic predicate using the predefined templates. This models the memory region related to this variable. Next, it checks whether this candidate predicate holds in the observed runs. If so, it uses this information to improve the analysis in the next iteration. Specifically, when it found a predicate, this typically describes only a part of the observed heap. This process is repeated until the all explored memory regions have been described. Finally, the candidate assertion is presented to the user.

Advantages and disadvantages. The main advantage of SLING is that it can infer shape properties, which many other tools do not support. And, it can infer properties in separation logic, as opposed to the usual predicate logic.

Some disadvantages of SLING include that it may infer actual behavior instead of intent and it relies on predefined templates. Moreover, finding assertions can take quite some time. In SLING's evaluation (Le et al., 2019) these times range from 10 s to over 14 min, this includes program execution, trace collection and invariant inference.

Finally, there are some features that limit the applicability of SLING. Many verifiers that use separation logic target concurrency and they use permissions to express access to variables. These permissions are often intertwined with the assertions because the specifications need to be self-framing. SLING, however, does not support concurrency nor does it infer permissions. SLING may also introduce existentially quantified variables which tend to be difficult for the backend solvers. Nonetheless, while these are some practical hurdles, the inferred specifications can be a good starting point for developers.

Our experience. We were unable to successfully install all dependencies of SLING, specifically LLDB v3.8 released in 2016, and thus we could not test the tool.

3.1.7. ALearner

ALearner (Pham et al., 2017) tries to infer specifications with limited test cases and limited usage of techniques such as symbolic execution. It specifically tries to infer assertions in methods. As input it takes a Java program and a set of test cases.

Given the input, ALearner first executes the test cases to collect program states. These program states are separated into failure cases and correct cases based on whether the test fails or runs successfully. Given such a set of program states, a classification algorithm is used to find a candidate assertion such that all cases are correctly classified. ALearner supports two classification algorithms. One is based on predefined templates, and boolean combinations of these, inspired by Daikon. Some of the predefined templates are `x*y = z`, `x != c` and `x > 0`. The second algorithm uses Support Vector Machine (SVM) to find assertions in the form $c_1x_1 + c_2x_2 + \cdots \geq k$. After the classification algorithm has been run, we are left with a candidate assertion. This assertion is often not correct due to the limited test set. Therefore, as a final step, ALearner uses active learning to improve the candidate assertion. The idea behind active learning is to generate new feature vectors (i.e. set of program states). The authors aim to generate feature vectors near the classification boundary. To achieve this, they select assertions closely related to the candidate assertion. It then takes the

program and enforces a program state corresponding to the selected assertion. Next, the test cases are run on the mutated program resulting in a new set of program states. This new set of program states is added to the previously collected set of program states. Finally, the classification algorithm can be used to find a new candidate assertion. This process is applied iteratively until the assertion converges.

Advantages and disadvantages. An advantage of ALearner is that its evaluation seems to indicate that it is capable of inferring better specifications with less test cases compared to Daikon thanks to the active learning approach. This active learning approach does introduce some additional overhead, on average it takes 40 s to find an assertion (Pham et al., 2017).

While the active learning approach of ALearner allows it to refine assertions in many cases and helps it overcome lacking test cases, it is still dependent on the availability of certain test cases as it is not capable of inferring sufficiently complex conditions. The active learning algorithm also relies on running the program to find new data points. This means that an incorrect program may lead to 'incorrect' program states and thus incorrect specifications.

Our experience. While the code is still available, we could not successfully run it on our examples.

3.1.8. DIG/SymInfer

Technique. *DIG* (Nguyen et al., 2014b), also known as SymInfer, is a tool focused on inferring invariants that describe a numerical relation between program variables, especially nonlinear invariants. This includes invariants such as `max(x,y) <= z - 4` and `q * y + r == x`.

To start, the user needs to mark the locations in the code for which they want to infer invariants. Given the code with target locations, DIG uses symbolic execution to obtain a set of symbolic states. It then uses several algorithms for inferring different kinds of invariants.

The first algorithm is used to find nonlinear equalities up to degree 2 (x^2). This algorithm generates terms, e.g. $1, a, a^2$ for all the available variables. These are combined with unknown coefficients to form an equality of the form $c_1 + c_2 * a + \cdots + c_x * y^2$. It then observes several concrete executions and uses these concrete values to solve the equation to get the coefficients. The found invariants are further refined in a loop where counterexamples are used to refute candidates and find new equations.

The second algorithm focuses on inferring inequality invariants. The algorithm enumerates octagonal terms, e.g. $x - y$ and $min(x, y, z)$. Then, it uses an SMT solver to find the smallest upper bound and largest lower bound for each term.

The third algorithm focuses on inferring invariants for arrays. It can find relations among array elements of the form $A = b_1 B^1 + \cdots + b_n B^n + c$ where A, B^i are distinct arrays whose elements are real-valued and b_i, c are coefficients. The approach is similar to the first algorithm, where several variables are created and the tool looks for equality relations among these variables. It can also find nested array relations, e.g. `A[i][j] == B[i+1][C[D[3j]]]`. To find nested array relations, it uses a similar approach to the second algorithm. It first enumerates possible array nestings. Then, it identifies relations among individual array elements using reachability analysis. Finally, this information is encoded into a satisfiability problem and solved using an SMT solver.

Finally, the tool post-processes the candidate invariants, removing violated and redundant ones.

Advantages and disadvantages. The main advantages of DIG is that it can infer nonlinear numerical invariants, which tend to be difficult for other tools. Because it focuses on this specific type of invariant, its applicability is limited as not all programs need nonlinear numerical invariants.

The main disadvantages is that it relies on the code for inference, meaning that it might infer invariants that are incorrect if the code is incorrect.

Our experience. DIG provides a Docker container with the necessary dependencies installed. Unfortunately, the Docker container does not support specification inference for Java at the time of writing. Thus, we were unable to successfully run DIG on our examples. We can however briefly reflect on our experience in preparing the examples as required for the tool. To use DIG, the user needs to manually indicate, in the code itself, where the invariants should be inferred. Specifically, the user defines an empty function which is called at the target location. The variables in scope at this location should be passed to this function as arguments as these are used by DIG to find invariants. In terms of user experience, this makes the code cluttered and this is not something one would like to have in a production environment.

3.2. Static techniques

This section discusses tools that use static approaches. This means that these tools do not execute the program for the specification inference process. This includes Houdini, Strongarm, KeY and "Mining preconditions of APIs in large-scale code corpus". Of these tools, we were only able to test Strongarm on the three prepared examples.

3.2.1. Houdini

Technique. *Houdini* (Flanagan and Leino, 2001) was originally developed for annotating legacy, unannotated programs. Given an unannotated program, Houdini first generates many candidate annotations. These candidates are generated based on heuristics. For example, it generates invariants that express that fields of a reference type cannot be null: `//@ invariant f != null`. It then uses ESC/Java to try and prove them. If a candidate cannot be proven to hold, it is discarded. Houdini can infer invariants, preconditions and postconditions.

Houdini has also been extended to guess specifications for libraries (Flanagan and Leino, 2001) in case the user does not have access to the source code of the library. The authors provide a pessimistic and optimistic approach for guessing specifications. The pessimistic approach makes assumptions such as "all pointers returned by library methods may be null" (Flanagan and Leino, 2001). This approach resulted in many false alarms. The optimistic approach makes assumptions such as "all pointers returned by library methods will be non-null" (Flanagan and Leino, 2001). As pointers may be null, this can cause Houdini to miss some runtime errors. Nonetheless, the optimistic approach was preferred by the authors as it could still detect other runtime errors.

Advantages and disadvantages. Houdini only reports assertions that could be proven if used to analyze a program. Moreover, it will report many of the simple properties that are likely to be true.

Houdini is limited in the kinds of properties it can report. It does not report disjunctions, numeric inequalities or properties such as: `\forall int i; 0 <= i <= expr ==> f[i] != null`. It may also report annotations that are subsumed by others such as `x != -1` and `x > 0`. Finally, Houdini's runtime can be quite long. The authors report a runtime of 62 h for 36,000 lines of Cobalt (Flanagan and Leino, 2001). They were, however, optimistic about improvements to this in the future.

Our experience. The original implementation of Houdini is no longer available. However, similar approaches have been implemented, e.g. in Boogie (Barnett et al., 2006) and GPUVerify (Betts et al., 2012). We have not evaluated these tools as they do not adhere to our selection criteria, i.e. they do not support inference of specifications for Java programs in a format usable by deductive verifiers.

3.2.2. Strongarm

Technique. *Strongarm* (Singleton et al., 2018) is a specification inference tool that specifically targets *strongest* postconditions. Given a precondition, which can be the default `true`, it computes the strongest postcondition. This strongest postcondition is then simplified thereby removing duplicate statements and tautologies from the postcondition.

Advantages and disadvantages. Unlike some other tools, Strongarm aims to minimize the size of the generated specifications. As a result, the inferred specifications are quite succinct which makes them nice to use for both users and tools.

The inferred specifications are relatively simple and Strongarm does not seem capable of inferring more complex specifications such as stating what the result of a search method over an array should be.

Our experience. While the code of Strongarm was merged into OpenJML (Cok, 2014), it seems to be currently unavailable for use[2]. Instead, we have used a virtual image with a pre-installed version of Strongarm that was provided by the authors.

We tried Strongarm both with the default precondition `true` as well as with the preconditions necessary for verification. We did not observe any significant differences between the generated specifications in these two cases.

Strongarm generated 60 annotations in total: 11 for the counter, 10 for the binary search and 39 for the arraylist. Of the 60 generated annotations, 23 of the annotations also occurred in the set of baseline annotations for verification. Thus 38% of the generated annotations were useful and necessary, covering 15% of the required annotations. It did not generate any incorrect specifications. All the other annotations it generated were correct but not required for verification in our baseline. A large part of the generated annotations consisted of behavior clauses and `also` keywords: 16 out of 60.

Strongarm is capable of inferring behavior clauses (which many other tools do not support), preconditions, postconditions and assignable clauses. Most of the generated specifications were comparisons between variables and/or values such as `requires 0 <= index` and `ensures \result == false`. It was also capable of inferring specifications that used built-ins such as `\old`, `\result` and `.length` for arrays. It was not able to find exceptional postconditions or more complicated postconditions that use quantifiers.

Unlike other tools, Strongarm is capable of distinguishing cases, derived from if-statements in the method. These are very helpful though sometimes a bit too precise. For example, in the generated annotations, it distinguished three separate cases in which the binary search returns -1:

- The length of the array is smaller than 1
- The value is larger than the last element in the array
- The value is smaller than the first element in the array

However, these cases can all be combined into one case (when the element is not in the array) thereby resulting in a clearer specification.

Strongarm is also capable of generating specifications with nested behavior clauses. While these may technically reduce the specification length (in terms of line numbers), we find that these make the specifications harder to understand. An example of a nested specification generated by Strongarm can be found in Listing 4.

3.2.3. KeY

Technique. There have been several projects about specification inference that implemented their technique in *KeY*.

The first project (Weiß, 2009) aims to infer loop invariants by combining symbolic execution with predicate abstraction. As input it expects the program, and optionally a set of predicates P. It then executes the loop symbolically, which results in a formula that represents the loop. Next, it approximates this formula by using predicate abstraction. It computes an abstraction of the formula using (conjunctions of) the elements in the provided finite set P. If no predicates are provided as input, the technique will use heuristics to derive some predicates. For example, it may use the postcondition to derive a loop invariant if it is a quantified statement over the same range as the loop. This process of computing an abstraction of the loop is repeated until a fix-point has been reached. This technique is only applied to loops without loop invariants.

The second project (Wasser, 2017) continues with the idea of combining symbolic execution and abstraction (Weiß, 2009). This work specifically moves towards better support for Java including non-standard control flows such as mutual recursion and break statements. To achieve this, they introduce new abstract domains for several types including arrays. This technique is then used to infer specifications for loops and recursive method calls.

The third project (Tabar et al., 2022), specifically focuses on inferring loop invariants that express the (absence of) data dependences. This is also an extension of the work by Weiß (2009). They symbolically execute a program to find an invariant candidate. Then, assuming this candidate, they execute the loop body once more. If the candidate holds at the end of the loop, then it is valid. If it was not valid, then there is an invariant from the beginning of the loop and a (different) invariant at the end of the loop. These will then be combined into the least common abstraction. This process is then repeated until a fix-point is reached. The candidates are always conjunctions of pre-defined predicates that capture atomic data dependences between memory areas.

Advantages and disadvantages. Tabar et al. (2022) is unique in that it focuses on finding loop invariants that express data dependences. The strength of Weiß (2009), Wasser (2017) is that it shows how to integrate symbolic execution and deductive verification in order to infer specifications. The technique can be applied to any program if suitable abstraction domains are available.

All of these approaches rely on symbolic execution of the program code to derive specifications. Because of this, the inferred specifications will reflect the actual program behavior, even if it is incorrect, instead of the intended behavior.

Our experience. All of the projects seem to be implemented as prototypes on branches of KeY. Unfortunately, the older implementations (Wasser, 2017; Weiß, 2009) have not been merged into the main development branch and therefore seem to be lost. There is also no publicly available artifact for (Tabar et al., 2022).

3.2.4. "Mining preconditions of APIs in large-scale code corpus"

Technique. Where many projects focus on analyzing the project for which you want specifications, Nguyen et al. (2014a) take a different approach. The main idea behind this work is that preconditions of APIs are expected to occur frequently in a large corpus whereas project-specific conditions are expected to occur less often. They analyze a large set of programs, all using the same API, to find preconditions for that API.

As input, one needs to indicate the API methods that need to be analyzed and provide projects that use this API. The tool first builds a control-flow graph for each method that calls the API. This graph is used to identify under which conditions an API method is called (control dependency analysis). This provides a precondition for each call site of the API method. As some of these are equivalent, the preconditions are normalized to combine equivalent conditions. When these are combined, the approach keeps track of each method that uses the equivalent precondition. As some preconditions might be implicit in the client code, the approach tries to infer some of these preconditions. They specifically try to address (1) the difference between strict and non-strict inequalities, (2) the difference between stronger conditions that imply weaker conditions, and (3) dynamic dispatch. Next, the preconditions are filtered to remove conditions that only occur once and conditions that are rarely checked before calling the API. Finally, the remaining preconditions are ranked into three separate lists, one for the receiver object, one for the arguments and one for combinations of them. The top-ranked preconditions are then reported to the user.

[2] A comment in the code seems to indicate that it is currently broken.

```
public class BinarySearchGood {

    /*+INFERRED
@      @ public normal_behavior
       @   {|
       @     requires !(value < sortedArray[0]);
       @     ensures \result == -1;
       @     {|
       @       requires (value > sortedArray[sortedArray.length - 1]);
       @     also
       @       requires !(0 <= sortedArray.length - 1);
       @       requires !(value > sortedArray[sortedArray.length - 2]);
       @     |}
       @   also
       @     requires (value < sortedArray[0]);
       @     ensures \result == -1;
       @   |}
       @*/
    public static int search(int[] sortedArray, int value) {
        ...
    }
}
```

Listing 4: Example showing nested specifications inferred by Strongarm for the binary search example.

Advantages and disadvantages. The main advantage, as well as disadvantage, is that this approach is used for a very specific type of program, namely an API for which a large set of usage examples is available. Depending on the project that you have, this can be very valuable, e.g. to find specifications for a library, or infeasible, e.g. when developing a new data structure.

It relies on the assumption that the APIs' preconditions occur frequently in the code where it is used. If an API is frequently misused, this may result in incorrect specifications.

Our experience. There does not seem to be a publicly available artifact for this project.

3.3. Natural language processing

The last category of tools that we discuss are tools that use natural language processing techniques. This includes Toradocu, also known as Jdoctor, C2S and ChatGPT. We have tested Toradocu and ChatGPT on the counter, binary search and arraylist example.

3.3.1. Toradocu/Jdoctor

Technique. Unlike many other tools, *Toradocu* (Blasi et al., 2018) derives specifications from programs' Javadoc documentation. It can generate preconditions for parameter values, postconditions and exceptional postconditions. The generated specifications are executable procedures as opposed to annotations.

Given code with Javadoc, Toradocu extracts the Javadoc tags `@param`, `@return` and `@throws` and the natural language comments that follow them. Next, it parses the natural language comments to identify the propositions (subject-predicate) pairs. These propositions are matched to a Java element based on pattern, lexical and semantic matching. Pattern matching can match common phrases such as "is positive" and "is not null" to corresponding Java expressions `>0` and `!= null`. Lexical matching matches a subject or predicate to similarly named code elements (based on Levenshtein distance). For example, in "if the comparator is null", "comparator" may be matched to the parameter whose type is `Comparator`. Semantic matching (based on Word Mover's Distance algorithm) is used to find words that are semantically similar. For example, "vertex" and "graph" are semantically related and Toradocu may use this to find matches that could not be found through lexical matching as the words are not similar. Then, it replaces each subject and predicate with matching Java code. Using this approach, Toradocu will produce a single translation for each Javadoc tag.

Note that Jdoctor is another name used for Toradocu, specifically for version 3.0. In the rest of this paper, we will simply refer to the tool as Toradocu.

Advantages and disadvantages. A big advantage of Toradocu is that it uses a source independent from the code to derive specifications. This provides a source that describes the intent of the programmer as opposed to the actual behavior of the program. This is of course only true if you are not using documentation generation techniques. Another advantage is that it allows users to write specifications in natural language instead of a formal specification language which is more difficult.

A disadvantage is that the comments may contain ambiguities which may lead to incorrectly translated specifications. Moreover, some of the original intent may be lost if the translation is inaccurate.

Another disadvantage is that Toradocu only generates translations for specific Javadoc tags. As a result, it does not generate specifications for, e.g. the effect of functions that do not return a value such as "enlarge" in our arraylist example.

As Toradocu produces executable procedures, some post-processing will be needed to obtain annotations from the generated output. However, as this is straightforward to do, we have included Toradocu in this list of tools.

Our experience. For our examples, we wrote documentation in the same style as the Javadoc for the List and ArrayList of Java 8. Unfortunately, Toradocu was not able to translate any of the documentation into executable specifications, i.e. it did not generate any specifications. After contacting the authors, this seems to be a limitation of the tool. They indicated that if we had written our documentation differently, Toradocu would have been able to translate it. For example, by changing:

```
//@throws IndexOutOfBoundsException if the index is out of
↪ range (index < 0 || index >= size())
```

into

```
//@throws IndexOutOfBoundsException if index < 0 || index
↪ >= size()
```

Toradocu can then generate the condition `args[0]<0` where `args[0]` refers to the `index` variable which is the first parameter of the method. This leads us to conclude that either the natural language processing would need to be improved, or one needs to be aware of what natural

language is suitable (i.e. can be translated by Toradocu) to use with this tool.

3.3.2. C2S

Technique. Similar to Toradocu, *C2S* (Zhai et al., 2020) uses documentation written in natural language to derive formal specifications. C2S generates pre- and postconditions for exceptional and normal behavior in JML based on natural language comments in the code.

Given a method with comments, C2S first extracts the natural language words. For these words, it looks up word-token pairs to find tokens that might represent it. These word-token pairs have been obtained by looking at natural language documentation (JDK documentation) and corresponding JML specifications.[3] The selected tokens are then assembled into a candidate specification based on grammar rules and the context of the method. Finally, developer test cases are used to filter the candidate specifications. This is achieved by turning the specifications into assertions that can be checked in the provided test cases. If an assertion is then violated, the specification is deemed invalid and thus will be discarded.

Advantages and disadvantages. C2S not only translates Javadoc but it also translates other natural language comments. This allows it to derive specifications for methods without a return value, unlike Toradocu. Moreover, its approach does not rely on matching specific patterns therefore it seems capable of translating a wide variety of natural language comments.

C2S uses a specification language grammar that describes the possible specification candidates. This grammar is relatively generic and therefore seems like a good approach. It takes into account keywords such as `\old`, `\result` and `.length` for arrays. However, this grammar does not express all possible specifications. For example, existentially quantified statements are not possible.

C2S uses test cases to filter specifications, therefore it can only derive specifications that match the observed behavior, which may not match the original intent.

Finally, one should reconsider using tools like C2S when using documentation generation techniques. If documentation is derived from code, and specifications are derived from this documentation, it is likely that the specifications will describe the actual behavior of the code instead of the original intent.

Our experience. There does not seem to be a publicly available artifact for C2S.

3.3.3. ChatGPT

Technique. ChatGPT (OpenAI, 2022) has been introduced at the end of 2022 and has gathered lots of attention. It is a large language model with which you can interact in a chat-like manner. This model has been trained using Reinforcement Learning from Human Feedback (RLHF). The approach is often explained as follows: "You can think of this as a very advanced autocomplete — the model processes your text prompt and tries to predict what's most likely to come next".[4]

Advantages and disadvantages. ChatGPT provides an easy user interface for specification inference. Moreover, it does not seem to be limited to specifications expressible in pre-defined templates such as tools like Daikon.

However, while it does not seem to be limited to pre-defined templates, it is unclear what the boundaries of ChatGPT's capabilities are. It should infer likely specifications based on other examples it has seen before. But it is not clear what was included in the training data set, and thus what examples it has been trained on. If one were to invent a new algorithm or a new type of specification, it is unknown how good the inferred specifications would be.

Similar to other tools, ChatGPT depends on the code to infer specifications. In most tools this introduces the possibility of inferring incorrect specifications if the code is incorrect. ChatGPT may also infer incorrect specifications, however, this can occur even when the code is *correct*.

Aside from depending on the code, the prompt that is given to ChatGPT also has significant influence on the results. As prompts have a big influence on the results, the field of prompt engineering has become popular. This field looks into how to optimally structure text to generate AI models. Some examples of approaches that seem to improve results include Chain-of-Thought prompting, where the large language model solves a problem in a series of steps, and self-refine, where the language model is prompted to reflect on its own solution and solve the problem again. For users it is important to look into these techniques to achieve optimal inference results.

Finally, we note that this is a very recent development. As such it may not be the most stable approach[5] and it may change significantly in the future.

Our experience. ChatGPT is a relatively quick and very easy-to-use tool. We have tested ChatGPT in several ways: (1) generating specifications from scratch, (2) translating documentation to formal specifications, and (3) generating from scratch with incorrect code.

First, we reflect on using ChatGPT to generate specifications when given program code as input and the prompt: "I have some Java code, can you annotate each function in the following examples with JML (Java Modeling Language) Specifications".

ChatGPT inferred a variety of specifications including preconditions, postconditions, invariants, loop invariants and assignable clauses. The expressions ranged from simple expressions such as `requires n >= 0` to nested expressions with quantifiers such as `ensures (\exists int i; 0 <= i && i <= \old(size); \old(list[i]) == o) ==> (size == \old(size) - 1) && (\forall int j; i <= j && j < \old(size); list[j] == \old(list[j + 1]));`. This specification was generated for the `remove` method of the Arraylist. It expresses that, if the element that should be removed was present in the list, then the size decreases by one and the elements after the removed element are all shifted a place to the left.

In total, ChatGPT (GPT-4) generated 42 annotations, 15 for the counter, 2 for the binary search and 25 for the arraylist. Of the generated annotations, 35 (83%) also occurred in our baseline specifications. This covered 24% of the baseline specifications. It generated 6 incorrect specifications, all for the arraylist example, which is around 14% of all generated specifications. It generated a single specification that was correct but not necessary.

Then, we used ChatGPT to translate English documentation to formal specifications. We used the following prompt combined with the program code: "Can you please translate the documentation in the following program to formal JML (Java Modeling Language) specifications?". The results were very similar to the results of the previous prompt. Moreover, we noted that ChatGPT did not strictly adhere to the prompt. For example, it generated the precondition `requires initialValue >= 0` even though this was not specified in the documentation. Similarly, it generated `pure` and loop invariants that were not described in the documentation. Other specifications that were generated were not clearly related to the documentation. For the `remove` method, described as "Removes the first occurrence of the specified object from the list", it generated the following specification: `!contains(o) || (contains(o) && size == \old(size)-1)`.

Next, we discuss ChatGPT's performance when given incorrect code and prompted to annotate each function with JML specifications. We introduced the following changes to our examples:

[3] https://www.cs.ucf.edu/~leavens/JML/examples.shtml

[4] Source: https://platform.openai.com/docs/quickstart/introduction.

[5] Some users claim to get worse results in newer versions with the same queries. It is unclear whether this is true.

Table 1
An overview of the inputs required for each tested tool as well as the average time needed for inference and the maximum reserved set size.

Tool name	Inputs	ArrayList		BinarySearch		Counter	
		Time	Max RSS	Time	Max RSS	Time	Max RSS
Daikon	Program and test	12 s	112 MB	9 s	97 MB	28 s	106 MB
EvoSpex	Program	2546 s	762 MB	96 s	105 MB	4 843 s	522 MB
Strongarm	Program and optionally preconditions	4 s	–	2 s	–	2 s	–
Toradocu/ Jdoctor	Program with Javadoc comments	41 s	1855 MB	4 s	185 MB	39 s	1851 MB
ChatGPT (4.0)	Program and prompt	.. s	–	.. s	–	.. s	–

- Renamed `decrByN(int n)` to `incrByN(int n)` in `Counter`
- Renamed `decr()` to `decrByN()` in `Counter`
- Changed implementation of `decrByN()` such that it always decreases the counter by 5 in `Counter` instead of decreasing by `n`
- Renamed the class `BinarySearchGood` to `RandomSearch`
- Renamed the `sortedArray` variable to `array` in `RandomSearch`
- Initialize the `size` variable to 18379138 in `ArrayList`
- Renamed `size()` to `ifjdsojo()` in `ArrayList`
- Renamed `contains(Object o)` to `notIn(Object o)` in `ArrayList`
- Renamed `enlarge()` to `aggrandize()` in `ArrayList`
- Renamed `add(Object o)` to `removeOne(Object o)` in `ArrayList`
- Renamed `isEmpty()` to `isNotEmpty()` in `ArrayList`

These changes were chosen as they introduce several situations such as (1) a situation where the implementation is correct but the name does not match, (2) an erroneous implementation, and (3) a situation where no information can be extracted from the names that are used, e.g. that an input array is sorted. The following changes were observed in ChatGPT's output:

- The implementation of `decr()` (used to be in `incr()`) was *modified* to decrease the counter and corresponding specifications were inferred.
- Correct specifications were inferred for the method that was renamed from `decr()` to `decrByN()`
- For the modified `decrByN()` method, where the implementation reduces the counter by 5, it inferred specifications that indicated that the counter was reduced by 5.
- For the `RandomSearch` example, it inferred correct specifications, including that the array was sorted.
- It correctly inferred specifications describing how the `size` variable is initialized in `ArrayList`
- It incorrectly places the start of a specification (`/*@`) at the beginning of the `ArrayList` class
- The other renames in `ArrayList` did not seem to influence ChatGPT as it inferred correct specifications for `ifjdsojo()`, `aggrandize()`, `removeOne(Object o)`, `notIn(Object o)`, `isNotEmpty()`

We observe that both implementation and the used names influence what specifications ChatGPT infers. Especially the implementation changes for the original `incr()` method were unexpected. When we asked ChatGPT to not modify the existing code, it did not modify the implementation of the original `incr()`, however it did rename a `decrByN()` method as there were two `decrByN()` methods resulting in invalid Java code. When asked not to modify the implementation for a second time, it also did not rename this method, however, it no longer inferred useful specifications for some methods. Specifically the `decrByN()` method had a pre- and postcondition that was `true`. The postcondition of `incrByN(int n)` became `true` as well. Thus, we advise users to carefully check whether ChatGPT modified the code to avoid introducing any errors.

Finally, we reflect on some general advantages and disadvantages of using ChatGPT. While many tools are limited in the type of specifications that they can infer, ChatGPT does not seem to share this limitation. We could generate preconditions, postconditions, assignable clauses, loop invariants and class invariants.

Another nice benefit of ChatGPT is that it can generate output in different specification languages. Many tools only support JML specifications, but we could also generate VerCors-specific annotations with ChatGPT.

One needs to be careful of how the prompt for ChatGPT is phrased. ChatGPT will try to follow the prompt precisely, and may not generate a type of specification if it is not requested. For example, if one asks ChatGPT to "annotate each function in the following examples with JML (Java Modeling Language) specifications", then it may not generate class invariants as these are not part of the specifications for functions.

We had significantly better results when using GPT-4 as opposed to GPT-3.5. When using GPT-3.5, the specifications were not correctly placed, e.g. a class invariant was placed inside a method. Moreover, the generated specifications were more simplistic, e.g. no exceptional postconditions and less quantified statements.

3.4. Summary of tested tools

Next, we provide a brief summary of the results of the tools that we tested on the three examples (counter, binary search and arraylist) namely Daikon, EvoSpex, Strongarm, Toradocu and ChatGPT.

Table 1 provides an overview of the requirements for using each tool. This includes all the inputs needed to run each tool, the time it took to infer specifications (average measured over 5 runs) as well as the maximum reserved set size (RSS) that was observed. Maximum RSS is the portion of memory that is used by a process in the main memory at one time. During the measurements the swap was turned off.

The maximum RSS could not be measured for Strongarm and ChatGPT as they could not be run as a separate process in a terminal. While we could not exactly measure the time ChatGPT took for inference, it took around 30 s for the complete answer to be revealed.

In terms of time, EvoSpex is a strong outlier which uses a lot more time for inference than the other tools. Toradocu uses the most memory in all examples. For both cases, one should take into account that the hardware on which ChatGPT is run is unknown. It is likely that it runs on a more powerful computer than these other tools which can be run on a laptop.

Aside from time and memory consumption of these tools, we also reflect on the quality of the inferred specifications in Table 2. We show how many specifications each tool generated, how many of these were correct (and could be found in our baseline), how many were correct but unnecessary (i.e. did not occur in our baseline), how many were incorrect, and the overlap of the generated specifications with the specifications we wrote for the examples.

Daikon generated the most specifications. EvoSpex and Strongarm naturally cover less of the baseline as they only generate postconditions. ChatGPT had the best performance on our examples: the generated specifications had the largest overlap with the baseline (24%). Moreover, it has the highest ratio of correct (and necessary) specifications (83%) compared to all specifications it generated. The other tools generate more specifications that are wrong and/or correct but not necessary for verification which will require more manual effort from the user to correct.

Table 2
An overview of the usefulness of the specifications generated with the tools we ran successfully compared to the baseline specifications.

Tool name	Total # of specifications	Correct	Unnecessary	Wrong	Coverage of baseline
Daikon	158	29 (18%)	47 (30%)	83 (53%)	19%
EvoSpex	25	8 (32%)	11 (44%)	6 (24%)	4%
Strongarm	60	23 (38%)	37 (62%)	0 (0%)	15%
Toradocu/Jdoctor	0	–	–	–	0%
ChatGPT (4.0)	42	35 (83%)	1 (2%)	6 (14%)	24%

4. Impact evaluation of tools

Given all of the tools that have been discussed, the natural questions that follow are "Which specifications cannot be generated yet?" and "Which parts of the specification writing process has the most room for improvement?". Therefore, in this section, we will investigate the impact of each tool on the specification writing process. We identify the types of specifications that can be generated with the current state-of-the-art specification inference tools. Moreover, we estimate the impact of each tool on the specification writing process. This allows us to (1) identify the types of specifications that cannot yet be generated, and (2) quantify the impact of each tool on the overall specification writing process.

For example, let us say we need 20 lines of specification to verify an example. How many, in terms of % of overall required, specifications can at most be generated by a tool? To estimate this impact, we will be using results from Lathouwers and Huisman (2022). In Lathouwers and Huisman (2022), a taxonomy was presented that identified the different types of specifications that are used for deductive verification in Java programs and they analyzed how often each type is expected to occur.

To identify the maximum impact of each tool, we first identify the type of specification that the tool infers. We then look for the smallest category in the taxonomy that includes all these specifications. If a tool infers multiple types of specifications, then we look for a smallest category for each type of specification that it infers. When we have found this category, we look up the corresponding 95% confidence interval in Lathouwers and Huisman (2022). These 95% confidence intervals was calculated as follows: given a set of 54 randomly chosen verified examples, the specifications were categorized according to the taxonomy (Lathouwers and Huisman, 2022). Each example is considered to be one data point, so it can reasonably be assumed that they are mutually independent. Next, the sample mean and sample standard deviation were calculated for each type of specification. We then apply the central limit theorem which allows us to assume a normal population distribution because the sample size is ≥ 30. Finally, the two-sided t-test with a significance level of 0.05 is used to calculate the 95% confidence interval.

Let us have a look at EvoSpex as an example of how to identify the maximum impact of a tool. EvoSpex generates postconditions. A category from the taxonomy that encompasses all possibly generated specifications is the category of postconditions, which has a 95% confidence interval of 18%–27%. This interval indicates that, of all specifications required to verify a Java program, 18 to 27% are expected to be postconditions. However, we can be even more specific by looking at a smaller category namely behavioral postconditions, meaning specifications that describe the behavior of a program *and* which are postconditions. The 95% confidence interval of this category is 14%–23%. As such, EvoSpex is expected to be able to generate *at most* 14 to 23% of the specifications required for the verification of a Java program.

The results of the impact analysis can be found in Table 3. For the tools that generate assertions, we take into account that these could potentially be used as pre- and postconditions with some manual work from the user. A visualization of the estimated maximum impact per tool can be found in Fig. 1. This visualizes the estimated *maximum* impact per tool, the actual impact may be lower than the visualized range.

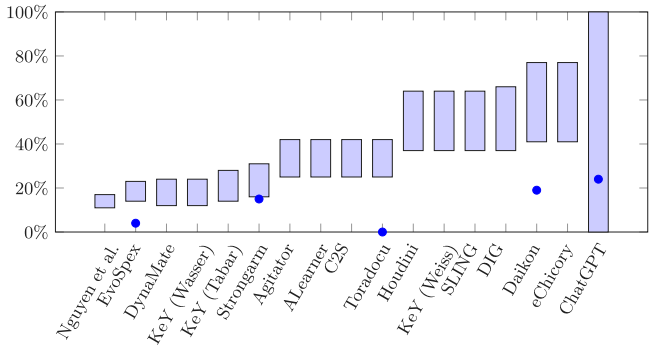

Fig. 1. A visualization of the estimated maximum impact range per tool. The dots indicate the observed coverage of the benchmark for the tools that were tested.

By comparing the type of specifications that the tools infer to the categories in the taxonomy for deductive verification specifications (Lathouwers and Huisman, 2022) we notice a few things. First, most tools target behavioral specifications, which are expected to make up between 47%–62% of all required specifications. Other categories such as proof assistance (9%–22%), permissions (9%–17%) and functions (5%–15%) are expected to make up significant portions as well but as of yet there is little support for the inference of such specifications. A handful of the tools mentioned in this survey support permission specifications, but only in the form of accessible and assignable clauses whereas verifiers such as VerCors and Verifast use different forms that are not supported yet.

Secondly, several tools focus on specific types of programs such a SLING, which focuses on heap-manipulating programs, and Tabar et al. (2022), who focus on loop invariants capturing data dependencies. These tools are only useful if applied in a situation where such specifications are expected to be used. However, users are unlikely to install a tool if it only supports a very specific type of specification. For these tools, it would be beneficial if they are distributed with other tools that support more generic specification inference.

Comparing the results from Tables 2 and 3, we see that, depending on the tool, there can be quite a big difference between the baseline covered in our examples and the maximum tool impact. For example, Daikon has an estimated maximum impact of 41%–77%, but only covered 19% of the baseline of our examples. Whereas, for Strongarm the estimated maximum impact and the impact on the baseline are very close, with a baseline coverage of 15% and an estimated maximum impact of 16%–31%. The examples chosen in our baseline are not a representative set of all possible verified programs and the results may therefore not be an accurate representation of each tool's capabilities. Nonetheless, none of tools cover more of the baseline than the estimated maximum impact.

Finally, we point out that, for some tools, the chosen category is quite large as a smaller category is not available in the current taxonomy as presented in Lathouwers and Huisman (2022). For example, Tabar et al. (2022) generate data dependence loop invariants. However, the data set does not contain information on data dependencies hence we have chosen the "Loop invariant" category. It is possible to get more accurate upper bounds, either by extending the taxonomy, data set and statistical analysis from Lathouwers and Huisman (2022) to include additional data, e.g. whether a specification is about data

Table 3
Overview of tools discussed in Section 3 with their expected maximum impact on the overall specification writing process.

Tool name	Type of specifications inferred	Corresponding taxonomy category/categories	Estimated maximum impact	Sum of impacts
Daikon (Ernst et al., 2007)	Preconditions, (normal) postconditions, loop invariants, invariants, modifies clauses, assertions	Behavioral preconditions Behavioral (normal) postconditions Behavioral loop invariants Behavioral (class) invariants Permissions on a method-level Behavioral assertions	11%–17% 14%–23% 12%–24% 1%–3% 3%–8% 0%–2%	41%–77%
eChicory (Alsaeed and Young, 2018)	Preconditions, (normal) postconditions, loop invariants, invariants, modifies clauses, assertions	Behavioral preconditions Behavioral (normal) postconditions Behavioral loop invariants Behavioral (class) invariants Permissions on a method-level Behavioral assertions	11%–17% 14%–23% 12%–24% 1%–3% 3%–8% 0%–2%	41%–77%
DynaMate (Galeotti et al., 2014)	Loop invariants	Behavioral loop invariants	12%–24%	12%–24%
Agitator (Boshernitsan et al., 2006)	Assertions	Behavioral assertions Behavioral preconditions Behavioral (normal) postconditions	0%–2% 11%–17% 14%–23%	25%–42%
EvoSpex (Molina et al., 2021)	Postconditions	Behavioral (normal) postconditions	14%–23%	14%–23%
DIG (Nguyen et al., 2017, 2014b)	Preconditions, normal postconditions, loop invariants and assertions	Behavioral preconditions Behavioral (normal) postconditions Behavioral loop invariants Behavioral assertions	11%–17% 14%–23% 12%–24% 0%–2%	37%–66%
ALearner (Pham et al., 2017)	Assertions	Behavioral assertions Behavioral preconditions Behavioral (normal) postconditions	0%–2% 11%–17% 14%–23%	25-42%
SLING (Le et al., 2019)	Preconditions, postconditions and loop invariants	Behavioral preconditions Behavioral (normal) postconditions Behavioral loop invariants	11%–17% 14%–23% 12%–24%	37%–64%
Houdini (Flanagan and Leino, 2001)	Preconditions, postconditions and loop invariants	Behavioral preconditions Behavioral (normal) postconditions Behavioral loop invariants	11%–17% 14%–23% 12%–24%	37-64%
Strongarm (Singleton et al., 2018)	Behavior clauses and strongest postconditions	Usability keywords Behavioral (normal) postconditions	2%–8% 14%–23%	16-31%
KeY (Wasser, 2017)	Loop invariants	Behavioral loop invariants	12%–24%	12-24%
KeY (Weiß, 2009)	Loop invariants and contracts for recursive method calls	Behavioral preconditions Behavioral (normal) postconditions Behavioral loop invariants	11%–17% 14%–23% 12%–24%	37-64%
KeY (Tabar et al., 2022)	Data dependence loop invariants	Loop invariants	14%–28%	14-28%
Nguyen et al. Nguyen et al. (2014a)	Preconditions of APIs	Behavioral preconditions	11%–17%	11-17%
Toradocu (Blasi et al., 2018)	Preconditions, (normal) postconditions and exceptional postconditions	Behavioral preconditions Behavioral (normal) postconditions Behavioral exceptional postconditions	11%–17% 14%–23% 0%–2%	25-42%
C2S (Zhai et al., 2020)	Preconditions, normal postconditions and exceptional postconditions	Behavioral preconditions Behavioral (normal) postconditions Behavioral exceptional postconditions	11%–17% 14%–23% 0%–2%	25-42%
ChatGPT (OpenAI, 2022)	Anything requested	All	0%–100%	0%–100%

dependencies, or doing a more in-depth evaluation of specification inference tools on a larger data set. In the case of loop invariants expressing data dependencies, an extension of the data set is required as none of the programs that are included in the data set use dependence predicates. Nonetheless, we believe the impacts are useful to gain a better understanding of the current state of the art and their impact on the overall specification writing process. For example, DynaMate generates loop invariants, and is therefore expected to generate at most 12%–24% of all specifications. In DynaMate's own evaluation (Galeotti et al., 2014), "it automatically built correctness proofs for 23 out of 26 subjects" ($\approx$ 88%). The estimated maximum impact of 12%–24% gives us a much better idea of DynaMate's impact on the complete specification writing process as opposed to its own evaluation that is focused on loop invariants. All without requiring any additional in-depth evaluation of the tool itself.

5. The ideal specification inference tool

Based on the tools we have discussed, our experience with inference tools, and the specifications required for deductive verification, we can discuss what an ideal annotation generator would look like. An ideal specification inference tool:

- Can generate output in multiple specification languages

- Can generate many types of specifications both in terms of what it describes (behavior, permissions, proof assistance) as well as the location (preconditions, postconditions, class invariants, loop invariants, etc.)
- Can automatically add the contracts to the code
- Does not require any changes to the original code to be used
- Does not require any manual intervention to rewrite or modify the generated annotations
- Does not require any hints about where to generate annotations
- Captures the intent of the code rather than the actual behavior
- Supports multiple inference techniques thereby enabling the use of different techniques depending on the program that is analyzed
- If supporting multiple inference techniques, it would help users to automatically detect what inference technique will likely provide the best results. Algorithm selection has been applied before in similar fields e.g. verification (Richter and Wehrheim, 2019) and SMT solving (Scott et al., 2021)
- While users can deal with some unnecessary or wrong annotations (Nimmer and Ernst, 2002), ideally most generated annotations are correct and useful for verification
- Generates human-readable annotations, i.e. it generates specifications in a similar format as the user would write it. This includes for example the use of behavior clauses and making specifications concise. Such human-readable annotations are essential to ensure that the user can understand the inferred specifications and possibly detect errors in them.
- Provide a supportive specification writing environment, similar to an IDE. Such an environment may include auto-completion, suggestions, as well as specification inference and specification checking, i.e. verification. Currently many specification inference tools are provided as separate tools from the verifiers. We believe this limits how often inference tools are used as users need to actively search for them. For users, it would be ideal if they can easily apply specification generation in the same environment as they use for verification without needing to install additional tooling.
- Allows for verifier-specific adjustments, e.g. by using specification features specific to the tool for better automation.

6. Related work

In this paper we have focused on tools that are available for Java programs and whose annotations are easily usable for deductive verifiers. In this section we discuss related work, as well as some other tools and techniques that have been proposed for specification inference but that did not meet the selection criteria for this survey.

6.1. Evaluations with Daikon

There are two notable evaluations for inference techniques, both of which focus on a comparison to Daikon.

The first evaluation, by Nimmer and Ernst (Nimmer and Ernst, 2002), compares Houdini and Daikon. As we were unable to install Houdini, this covers a gap that our research was unable to address. Overall users seemed to prefer Daikon over Houdini. The study also presents some practical recommendations such as

- Hiding annotations confused users, therefore they recommend being able to hide/show annotations
- Users feel hindered when the inference takes too long
- While Daikon generated several incorrect invariants, this did not seem to have a significant impact on the users

The second study (Polikarpova et al., 2009) compared contracts written by developers to contracts inferred by Daikon. They have shown that 90% of the inferred specifications were correct, and 64% were relevant. They have also shown that inference tools can generate many more specifications that are both correct and interesting. Moreover, they show that the tools infer around 59% of the contracts written by developers. This study specifically used Daikon to infer contracts for Eiffel code. In our examples the correct and relevant ratio of generated specifications is significantly lower with 19%. However, the study also notes that the percentage of relevant assertions vary widely between programs. Our evaluation of the tools is limited as it has only been applied to three examples. As future work, it would be interesting to do a more in-depth evaluation with a large representative set of programs.

While both of these studies provide a comparison to Daikon, they do not provide an overview of the specification inference field as we have done in our work. Nonetheless, their conclusions provide valuable insight into specification inference and its application in practice.

6.2. Loop invariants

Over the years, the inference of loop invariants has garnered much attention. Not only for deductive verification but also for other verification communities such as model checking. As such, many tools have been proposed that use a range of techniques. While we have included the tools for Java programs in our survey, there is more to explore. We refer the interested reader to the survey by Furia et al. (2014) which dives into the fundamental patterns that can be found in loop invariants. In this section we briefly discuss recently proposed tools for loop invariant generation that do not support Java programs. This includes Code2Inv, DySy, PIE/LoopInvGen, Vampire, PILAT and Aligator.jl.

Si et al. (2018) propose a reinforcement learning-based approach. This was implemented in Code2Inv and evaluated on SyGuS competition problems. They use a graph neural network model to represent the external memory of a program. Next, it queries this model with an attention mechanism thereby incrementally constructing a loop invariant. As a result, it is more suited to loop invariants in the conjunctive normal form as opposed to disjunctive properties.

Csallner et al. (2008) proposed the DySy tool as an improvement on the dynamic inference of Daikon. This approach is also mentioned in the survey by Furia et al. (2014). They combine concrete execution of test cases with symbolic execution of the same tests. They observe that DySy can infer a large part of the interesting invariants that are inferred by Daikon, while generating less of the irrelevant invariants.

Padhi et al. (2016) focus on generating loop invariants that are provably correct for a program with respect to a given specification. It does not depend on predefined templates, and thereby can generate more expressive invariants than some other approaches. Instead of using predefined templates, it uses a program synthesizer to generate atomic predicates. Given a set of good and bad program states, it learns an invariant which is a combination of atomic predicates, such that it satisfies good states and falsifies bad states. As such, we can see clear commonalities with the approach of ALearner. When an invariant has been found, it is checked whether it is sufficient to prove the program correct. If not, the counterexample is used to further refine the invariant. As it does not rely on predefined templates, it uses a program synthesizer to generate the atomic predicates.

Ahrendt et al. (2015) shows how the first-order theorem prover Vampire can be used to generate quantified invariants of loops with arrays. They provide a general framework for how this can be used, and thus can be applied to many different programming languages. While they claim to have integrated it into KeY, which does support Java, this integration does not seem to be available. Therefore, we have chosen to include it in this section of the paper instead. Instead of using this to generate loop invariants, it is also proposed to use this as an improved way for reasoning about loops, one that does not require user guidance. While it can generate loop invariants for the user, the intended way seems to be to delegate part of the proof to Vampire. They combine symbolic elimination and consequence finding,

techniques from saturation theorem proving. One of the unique features of this technique is that it supports the generation of invariants with quantifier alternations.

de Oliveira et al. (2016) focus on generating polynomial invariants for C programs with PILAT using linear algebra theory. Their approach first reduces the analysis of solvable loops to the analysis of linear loops. Then, it generates inductive invariants for the linear loops using polynomial complexity linear algebra algorithms.

Humenberger et al. (2018) propose Aligator.jl, another approach for generating polynomial invariants, for programs written in Julia. They translate a program into a system of algebraic recurrences. This system of recurrences is then solved using techniques from symbolic computation.

As a final note on loop invariants, while most tools rely on loop invariants, there is a different approach namely loop contracts (Ernst, 2022). A loop contract consists of a precondition and a relational postcondition. As loop contracts have received little attention so far, it seems like inference techniques for such contracts have not been proposed yet. It would be interesting to investigate whether techniques for inference of regular pre- and postconditions can be reused to infer loop contracts.

6.3. Permission annotations

Several tools have been proposed that infer some kind of access permission annotations. The two that we will discuss briefly are Sample (Dohrau et al., 2018) and Sip4J (Sadiq et al., 2019).

Sample infers access permission annotations for Viper (Müller et al., 2016) programs. Viper is used as a back-end for several deductive verifiers. Sample can infer access permissions for array programs, though it does not yet support reasoning about dynamically created objects. As the annotations are inferred in Viper's own language, Sample cannot be directly applied to the verifiers that use Viper as a back-end and translating them to the front-end is not always straightforward.

Sip4J focuses on inference of access permission contracts. These contracts can be used to identify which methods in a sequential program could be executed in parallel. The access permissions that they are interested in include `unique`, `immutable`, `full`, `share` and `pure`. These are written in the form of Plural annotations which can be used to "enforce typestate-based protocols using permissions" (Bierhoff and Aldrich, 2008). However, these annotations are not immediately usable by general deductive verifiers such as OpenJML (Cok, 2014) and VerCors (Blom et al., 2017) as they use a different notion of permission annotations.

Experience shows that separation logic verifiers often require more annotations because users need to express access permissions. As such, it is important to further investigate permission inference, especially for front-end languages.

6.4. Separation logic annotations

In this section we briefly touch on approaches that specifically target specifications for separation logic-based verifiers such as VerCors (Blom et al., 2017) and Verifast (Jacobs et al., 2011).

One of the techniques that we did not discuss so far is shape analysis. Shape analysis is a type of pointer analysis that looks for properties of data structures. Over the years several tools have been proposed that used shape analysis to e.g. discover pre- and postconditions (Calcagno et al., 2011; Le et al., 2014) as well as predicates (Boockmann and Lüttgen, 2020). These specification can be used to prove memory-safety with separation-logic based verifiers.

The work by Luo et al. (2010) proposes a top-down abductive approach as an alternative to the bottom-down abductive approach from Calcagno et al. (2011). They want to infer specifications for some unknown procedure call in a program. To find a precondition, they analyze the code before the call to the unknown procedure. And, to find a postcondition, they analyze the code after the call to the unknown procedure. The idea thus being to use the context to derive sufficient specifications.

Another approach for inference of separation logic specifications, was proposed by Vogels et al. (2011). Vogels et al. proposed techniques to infer (1) automatically open and close predicates and (2) automatically apply lemmas. Both of which, like the shape analysis, can be used for the verification of memory-safety. It is currently unclear whether Verifast supports annotation inference for Java programs. The shape analysis is only available for C programs at the time of writing.

They also proposed Automated Verifast (Mohsen and Jacobs, 2016) which generates predicates and automatically tries to fix verification failures. The techniques used by Automated Verifast are not clearly described and unfortunately the implementation was never incorporated into Verifast.

6.5. Additional inference techniques

Next, we briefly discuss two unique approaches that we did not get to discuss due to our selection criteria.

The first approach (Srivastava and Gulwani, 2009) aims to derive specifications with quantifiers, specifically $\forall/\forall\exists$-quantified invariants. They combine templates with predicate abstraction to find the invariants. As many tools still seem to generate a limited amount of quantified statements, it would be interesting to take the templates from this approach and integrate them into other tools.

The second approach (Alshnakat et al., 2020) investigates how Horn clause solvers, typically used for model checking, can be used to infer specifications for deductive verification. They have shown that the prototype could infer required pre- and postconditions. While the current approach is still limited, e.g. no loop invariants, arrays or heap-allocated data structures, the technique seems promising and would be interesting for future work.

6.6. Metatools

Aside tools implementing inference techniques, there are also tools to improve or help to apply such tools. We refer to these tools as metatools, meaning tools that are supposed to be used in combination with another tool, e.g. to improve the efficiency of a technique. Two such metatools are AIMS and DeltaSpec, both of which are used together with SpecFuzzer (Molina et al., 2022), a tool for the inference of specifications based on grammar-based fuzzing, dynamic invariant detection with Daikon and mutation analysis.

AIMS (Garg et al., 2023) is a tool to improve the efficiency of inference techniques that use mutants from mutation testing. Given a set of mutants, it selects a (sub)set that is well-suited for specification inference. By selecting a smaller set for inference, less mutants need to be executed, thereby reducing the number of computations. Combining this approach with SpecFuzzer, they could infer assertions for examples that previously timed out.

Another metatool is DeltaSpec (Degiovanni et al., 2023), which generates specifications on a commit-level. Given the code before a commit and after a commit, DeltaSpec tries to generate specifications that capture the changes of the commit. Such a tool can help to maintain specifications while a codebase evolves.

These approaches are complementary to the specification inference techniques discussed in this paper. While they do not solve the inference problem, they can help to make the inference more efficient or make it easier to maintain specifications.

6.7. Translating other formalisms to and from JML

In this survey we have focused on using various artifacts, e.g. documentation and tests, to infer new specifications. However, another interesting approach is to translate other formalisms, such as automata,

to obtain specifications. Over the years, many such translations have been proposed, especially for translating to and from JML. Some examples include the translation from B machines to JML (Cataño et al., 2012), OCL to/from JML (Hamie, 2004), JML to executable Java (Beckert et al., 2020), Alloy expressions to JML (Grunwald et al., 2014), JML from temporal properties (Giorgetti and Groslambert, 2006), JML from VDM-SL (Tran-Jørgensen et al., 2018), JML from security automata (Huisman and Tamalet, 2009), desugaring JML (Raghavan and Leavens, 2005), as well as translating between JML-like specification languages (Armborst et al., 2024).

Unlike many approaches discussed in this survey, these techniques focus on translating various formalisms, as opposed to generating new specifications. While these are very useful, e.g. for reusing formal artifacts, we believe they will have a limited impact on reducing the specification bottleneck as they require another formal artifact. So, while they are perhaps similar to Toradocu and C2S, discussed in Section 3.3, which translate between natural language and JML, we think those can be more effective at reducing the specification bottleneck as they do not require another formalization.

6.8. Impact of programming language on specifications

Different programming languages provide different correctness guarantees through, e.g. the type system. A perfect example of this is the Rust language whose type system ensures memory safety. Astrauskas et al. (2019) have encoded Rust's type system thereby simplifying the verification of Rust programs. Specifically, they use permissions to encode Rust's type system. As these are encoded automatically, the user does not need to write these specifications anymore.

A somewhat similar approach is presented by Fiala et al. (2023). They present an approach for the synthesis of programs in safe Rust. Like Astrauskas et al. they can use simpler specifications by leveraging Rust's type system.

We use these examples to point out the influence of the programming language on the specifications as using a different language may require the user to write less or additional specifications. For example, for Java we typically have to provide permission specifications to prove memory safety with VerCors. As this survey was focused on Java programs, it is possible that we have not discussed tools that infer specifications that are not required for Java.

7. Challenges

While Section 5 discussed practical recommendations for the development of inference tools, this section touches on some research challenges for specification inference.

Investigate inference of other specification types Nearly all tools that we discussed in this paper are focused on the generation of behavioral specifications, i.e. specifications that describe the functional behavior of the program. However, Lathouwers and Huisman (2022) has shown that these are expected to only make up between 47%–62% of all required specifications. For future work it would therefore be interesting to focus on other types of specifications such as predicates (Boockmann et al., 2018), permissions (Dohrau et al., 2018) and proof assistance specifications. Especially proof assistance specifications seem difficult, though there has been some work on the generation of lemmas (Johansson, 2019; Singher and Itzhaky, 2021) and the automatic (un)folding of predicates (Vogels et al., 2011).

Investigate other truth sources Most of the proposed techniques only use the code and possibly a test suite to derive specifications. It would be interesting to further investigate other possible truth sources such as documentation, usage of code and similar types of programs. One can also consider including the user as a truth source by developing interactive inference techniques. For example, by asking the user questions about the intended behavior of the program and deriving specifications based on the user's answers. While this does introduce some manual effort into the process, it can avoid the pitfall of deriving specifications that describe the actual instead of the intended behavior of the program. We believe it can still be beneficial if it requires less effort compared to manually writing specifications and if the specifications it produces are useful and relevant.

Use inference techniques for other verifiers Many tools have been proposed for specification inference that never present the specifications to the user. For example, DIDUCE (Hangal and Lam, 2002) generates invariants and immediately checks the program using them. In a similar manner, many tools have been proposed that generate invariants that are used by model checkers. It is unclear whether the techniques applied in these tools can (1) generate specifications that are useful for deductive verification, and (2) whether the generated specifications are easy to read and understand by users. Even if they are not easy to read and understand, perhaps such techniques can be combined with rewriting tools that rewrite specifications into more human-readable formats.

8. Conclusion

In this paper, we have presented an overview of specification inference tools that are available for deductive verification of Java programs. As such, we explained the underlying techniques that are used for specification inference. Moreover, we discuss our experience using these tools where possible. Based on the type of specifications that each tool infers, we have made an estimate of the tools' impact on the overall specification writing process. Using this data and experience, we described what an ideal specification inference tool would look like. This includes e.g. being able to generate output in multiple specification languages, generating many different types of specifications using one tool, and generating human-readable annotations. Finally, we identified interesting areas for future research into specification inference, such as investigating what other artifacts or sources can be used for specification inference. For future work, it would be interesting to extend this survey to include specification inference tools that target other programming languages for an even more complete overview.

CRediT authorship contribution statement

Sophie Lathouwers: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Marieke Huisman:** Conceptualization, Funding acquisition, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Sophie Lathouwers reports financial support was provided by Dutch Research Council. Marieke Huisman reports financial support was provided by Dutch Research Council.

Data availability

The data for this paper is available at https://doi.org/10.4121/9c83933e-8406-4e49-ac4d-1f8bb55ed988. This includes the examples, as well as any additional artifacts such as tests and documentation, that were used to test tools that generate specifications. It also provides Virtual Machines that enable people to use Daikon, EvoSpex, Strongarm and Toradocu. ChatGPT could not be redistributed for replication as it is a proprietary service.

Acknowledgments

This research was supported by a grant from the Dutch Research Council (NWO): VICI 639.023.710 Mercedes project.

Appendix A. Examples used to test specification inference tools

This appendix provides the program code that was used as input for the various specification inference tools. This appendix shows the examples that we have verified with OpenJML (v0.17.0-alpha-14). The program code (without the specifications) has been used as input to test various specification inference tools (see Section 3). Three examples were used: a counter (see Listing 5), a binary search (see Listing 6) and an arraylist (see Listing 7).

```
public class Counter {
  public int value;

  //@ ensures this.value == 0;
  public Counter() {
    value = 0;
  }

  //@ requires Integer.MIN_VALUE <= initialValue && initialValue <= Integer.MAX_VALUE;
  //@ ensures this.value == initialValue;
  public Counter(int initialValue) {
    value = initialValue;
  }

  //@ requires Integer.MIN_VALUE <= this.value && this.value < Integer.MAX_VALUE;
  //@ assignable value;
  //@ ensures this.value == \old(this.value) + 1;
  public void incr() {
    value = value + 1;
  }

  //@ requires Integer.MIN_VALUE <= this.value && this.value <= Integer.MAX_VALUE - n;
  //@ requires n >= 0;
  //@ assignable this.value;
  //@ ensures this.value == \old(this.value) + n;
  public void incrByN(int n) {
    int tmp = n;
    //@ loop_invariant this.value == \old(this.value) + (n-tmp);
    //@ decreases tmp;
    while (tmp > 0) {
      incr();
      tmp = tmp - 1;
    }
  }

  //@ requires Integer.MIN_VALUE + 1 <= this.value && this.value <= Integer.MAX_VALUE;
  //@ assignable this.value;
  //@ ensures this.value == \old(this.value) - 1;
  public void decr() {
    value = value - 1;
  }

  //@ requires Integer.MIN_VALUE + n <= this.value && this.value <= Integer.MAX_VALUE;
  //@ requires n >= 0;
  //@ assignable this.value;
  //@ ensures this.value == \old(this.value) - n;
  public void decrByN(int n) {
    int tmp = 0;
    //@ loop_invariant this.value == \old(this.value) - tmp;
    //@ decreases n-tmp;
    while (tmp < n) {
      decr();
      tmp = tmp + 1;
    }
  }

  //@ ensures \result == this.value;
  //@ pure
  public int get() {
    return value;
  }

  //@ requires Integer.MIN_VALUE <= n && n <= Integer.MAX_VALUE;
  //@ assignable this.value;
  //@ ensures this.value == n;
```

```
  public void set(int n) {
    value = n;
  }
}
```

Listing 5: Counter program that has been verified using OpenJML. Adapted from a VerCors example: https://github.com/utwente-fmt/vercors/blob/82ffdf3688c63a1d52dffb93d154e16b746fe47a/examples/concepts/permissions/Counter.java.

```
public class BinarySearchGood {

    //@ requires sortedArray != null;
    //@ requires 0 < sortedArray.length < Integer.MAX_VALUE;
    //@ requires \forall int i,j; 0 <= i < j < sortedArray.length; sortedArray[i] <=
    ↪ sortedArray[j];
    //@ ensures 0 <= \result < sortedArray.length <==> (\exists int i; 0 <= i <
    ↪ sortedArray.length; sortedArray[i] == value);
    //@ ensures \result != -1 ==> sortedArray[\result] == value;
    //@ ensures \result == -1 <==> (\forall int i; 0 <= i < sortedArray.length; sortedArray[i] !=
    ↪ value);
    //@ pure
    public static int search(int[] sortedArray, int value) {
        if (value < sortedArray[0]) return -1;
        if (value > sortedArray[sortedArray.length-1]) return -1;
        int lo = 0;
        int hi = sortedArray.length-1;
        //@ loop_invariant 0 <= lo < sortedArray.length;
        //@ loop_invariant 0 <= hi < sortedArray.length;
        //@ loop_invariant (\exists int i; 0 <= i < sortedArray.length; \old(sortedArray[i]) ==
    ↪ value) ==> sortedArray[lo] <= value <= sortedArray[hi];
        //@ loop_invariant \forall int i; 0 <= i < lo; sortedArray[i] < value;
        //@ loop_invariant \forall int i; hi < i < sortedArray.length; value < sortedArray[i];
        //@ loop_decreases hi - lo;
        while (lo <= hi) {
            int mid = lo + (hi-lo)/2;
            if (sortedArray[mid] == value) {
                return mid;
            } else if (sortedArray[mid] < value) {
                lo = mid+1;
            } else {
                hi = mid-1;
            }
        }
        return -1;
    }
}
```

Listing 6: Binary search program that has been verified using OpenJML. Source: https://www.openjml.org/examples/bubble-sort.html

```
public class ArrayList {
  //@ public invariant -1 <= size <= Integer.MAX_VALUE;
  //@ public invariant list != null;
  //@ public invariant size < list.length;
  //@ public invariant (\forall int i; 0 <= i <= size; list[i] != null);
  public int size;
  //@ public invariant 0 <= list.length <= Integer.MAX_VALUE;
  public Object[] list;

  //@ ensures size == -1;
  //@ ensures \fresh(list);
  //@ pure
  public ArrayList() {
    size = -1;
    list = new Object[10];
  }

  //@ ensures \result == size;
  //@ pure
  public int size() {
    return size;
  }

```

```
  //@ public normal_behavior
  //@   requires 0 <= index && index <= size;
  //@   assignable \nothing;
  //@   ensures \result == list[index];
  //@ also
  //@ public exceptional_behavior
  //@  requires index < 0 || index > size;
  //@  assignable \nothing;
  //@  signals (IndexOutOfBoundsException) true;
  public Object get(int index) {
    if (0 <= index && index <= size) {
      return list[index];
    } else {
      throw new IndexOutOfBoundsException();
    }
  }

  //@ requires o != null;
  //@ ensures \result == false <==> size == -1 || (\forall int i; 0 <= i <= size; list[i] != o);
  //@ ensures \result == true <==> (\exists int i; 0 <= i <= size; list[i] == o);
  //@ pure
  public boolean contains(Object o) {
    //@ loop_invariant 0 <= i <= size+1;
    //@ loop_invariant (\forall int j; 0 <= j < i; list[j] != o);
    //@ decreases size - i;
    for (int i = 0; i <= size; i++) {
      if (list[i] == o) {
        return true;
      }
    }
    return false;
  }

  //@ public normal_behavior
  //@   requires 0 <= index <= size;
  //@   requires o != null;
  //@   requires \type(Object) <: \elemtype(\typeof(list)); // needed to prevent
    ↪ PossiblyBadArrayAssignment warning
  //@   assignable list[index];
  //@   ensures list[index] == o;
  //@ also
  //@ public exceptional_behavior
  //@   requires index < 0 || index > size;
  //@   assignable \nothing;
  //@   signals (IndexOutOfBoundsException) true;
  public void set(int index, Object o) {
    if (0 <= index && index <= size) {
      list[index] = o;
    } else {
      throw new IndexOutOfBoundsException();
    }
  }

  //@ private normal_behavior
  //@   requires list.length < Integer.MAX_VALUE;
  //@   requires \type(Object) <: \elemtype(\typeof(list));
  //@   assignable list;
  //@   ensures \fresh(list);
  //@   ensures \type(Object) <: \elemtype(\typeof(list));
  //@   ensures (\forall int i; 0 <= i <= size; list[i] == \old(list[i]));
  //@   ensures list.length > \old(list.length);
  private void enlarge() {
    Object[] newList;
    if (list.length >= Integer.MAX_VALUE-10 && list.length < Integer.MAX_VALUE) {
      newList = new Object[Integer.MAX_VALUE];
    } else {
      newList = new Object[list.length + 10];
    }
    //@ loop_invariant 0 <= i <= size + 1;
    //@ loop_invariant (\forall int j; 0 <= j < i; newList[j] == \pre(list[j]));
    //@ decreases size - i;
```

```
    for (int i = 0; i <= size; i++) {
      newList[i] = list[i];
    }
    list = newList;
  }

  //@ public normal_behavior
  //@   requires size < Integer.MAX_VALUE - 1;
  //@   requires o != null;
  //@   requires \type(Object) <: \elemtype(\typeof(list));
  //@   assignable size, list, list[*];
  //@   ensures (\forall int i; 0 <= i <= \old(size); list[i] == \old(list[i]));
  //@   ensures size == \old(size) + 1;
  //@   ensures list[size] == o;
  //@ also
  //@ public exceptional_behavior
  //@   requires size == Integer.MAX_VALUE - 1;
  //@   assignable size;
  //@   signals (IndexOutOfBoundsException) true;
  public void add(Object o) {
    if (size == Integer.MAX_VALUE - 1) {
      throw new IndexOutOfBoundsException();
    }
    if (size == list.length - 1) {
      enlarge();
    }
    size = size + 1;
    list[size] = o;
  }

  //@ public normal_behavior
  //@   requires o != null && size >= 0;
  //@   requires (\exists int i; 0 <= i <= size; list[i] == o); // The element to remove is in the
    ↪ list
  //@   assignable list[*], size;
  //@   ensures size == \old(size) - 1;
  //@   ensures (\exists int i; 0 <= i <= \old(size); \old(list[i]) == o && (\forall int j; 0 <= j
    ↪ < i; list[j] == \old(list[j])) && (\forall int k; i <= k <= size; list[k] ==
    ↪ \old(list[k+1])));
  //@ also
  //@ public normal_behavior
  //@   requires o != null && size >= 0;
  //@   requires (\forall int i; 0 <= i <= size; list[i] != o); // The element to remove is NOT in
    ↪ the list
  //@   assignable list[*], size;
  //@   ensures size == \old(size);
  //@   ensures (\forall int i; 0 <= i <= size; list[i] == \old(list[i]));
  //@ also
  //@ public normal_behavior
  //@   requires size == -1; // Empty list
  //@   assignable \nothing;
  public void remove(Object o) {

    //@ loop_invariant 0 <= i <= \old(size) + 1;
    //@ loop_invariant (\forall int k; 0 <= k <= size; list[k] == \old(list[k]));
    //@ loop_invariant size == \old(size);
    //@ loop_invariant (\forall int k; 0 <= k < i; list[k] != o);
    //@ loop_modifies list[*], size;
    //@ decreases size - i;
    for (int i = 0; i <= size; i++) {
      if (list[i] == o) {
        //@ loop_invariant i <= j <= size;
        //@ loop_invariant (\forall int k; 0 <= k < i; list[k] != o);
        //@ loop_invariant (\forall int k; 0 <= k < i; list[k] == \old(list[k]));
        //@ loop_invariant (\forall int k; i <= k < j; list[k] == \old(list[k+1]));
        //@ loop_invariant (\forall int k; j < k <= size; list[k] == \old(list[k]));
        //@ loop_modifies list[j..size];
        //@ decreases size - j;
        for (int j = i; j < size; j++) {
          list[j] = list[j+1];
        }
```

```
161           size = size - 1;
162           return;
163         }
164       }
165     }
166
167     //@ ensures \result <==> size == -1;
168     //@ pure
169     public boolean isEmpty() {
170       return size == -1;
171     }
172
173
174 }
```

Listing 7: Arraylist program that has been verified using OpenJML.

Appendix B. Overview of specification inference tools

This appendix provides an overview of the tools that are discussed in Section 3. The overview can be found in Table 4. For each tool we show what needs to be provided as input, briefly describe the methodology, mention what type of specifications the tool can infer and whether the tool is openly available.

Table 4
Overview of tools for specification inference for deductive verification of Java programs. For each tool we mention the required input for the tool, briefly describe the methodology, mention what type of specifications it generates and note whether the tool is openly available. ✓indicates that the tool is available and could be run on our own examples. 'o' indicates that the tool is available but we were unable to run it on our own examples, e.g. because we could not successfully compile the tool. '-' indicates that the tool is not available.

Tool	Input	Methodology	Output	Artifact available?
Daikon (Ernst et al., 2007)	Code and tests or data trace files	Observes program runs and reports properties that were true over the observed executions.	Preconditions Postconditions Invariants	✓
eChicory (Alsaeed and Young, 2018)	Code and tests or data trace files	Extension of Daikon that can infer invariants about objects in dynamic relationships such as MVC.	Invariants	o
DynaMate (Galeotti et al., 2014)	Code annotated with pre- and postconditions	Generates tests with EvoSuite and then uses two dynamic invariant detectors (Daikon and Gin-Dyn). The tests are used to discard invalidated candidates.	Loop invariants	o
Agitator (Boshernitsan et al., 2006)	Code and partial test suite	Combination of test-input generation and dynamic invariant detection	Assertions	–
EvoSpex (Molina et al., 2021)	Code and user-defined ranges for test set	It generates valid and invalid pre/post pairs. Then, it uses a genetic algorithm that satisfy the valid pairs and leaves out the invalid pairs.	Postconditions	✓
SLING (Le et al., 2019)	Code, program location, set of predefined predicates defining data structures and a set of sample inputs	While the program is running, they capture memory information of the variables at the provided program location. This is used to instantiate predicate parameters with boundary values. Then they discard statements that cannot be proven.	Preconditions Postconditions Loop invariants	o
ALearner (Pham et al., 2017)	Instrumented code and optional test cases	They divide the program states into two groups. They then generate assertions based on templates from Daikon, specifically looking for a Boolean combination of assertions that separates these two states perfectly.	Assertions	o
DIG (Nguyen et al., 2014b)/SymInfer (Nguyen et al., 2017)	Code	Analyze program traces to infer polynomial and array invariants	Invariants	o
Houdini (Flanagan and Leino, 2001)	Code	Generates candidate annotations based on heuristics and removes the ones that cannot be proven.	Preconditions Postconditions Invariants	–
Strongarm (Singleton et al., 2018)	Code and optional precondition(s)	Uses symbolic execution with predicate transformers to infer strongest postconditions.	Postconditions	✓

(continued on next page)

Table 4 (*continued*).

Tool	Input	Methodology	Output	Artifact available?
KeY (Weiß, 2009; Wasser, 2017; Tabar et al., 2022)	1) Code and optionally predicates 2) Code 3) Code	1) Combines symbolic execution and predicate abstraction 2) Combines symbolic execution and abstract interpretation 3) Combines symbolic execution and predicate abstraction	Loop invariants	–
No toolname (Nguyen et al., 2014a)	Client code and API	Computes control dependence relation from client call sites invoking APIs. These are used to mine potential conditions used to reach the call sites. These conditions are used to infer preconditions for each API.	Preconditions	–
Toradocu/Jdoctor (Blasi et al., 2018)	Code with Javadoc comments	Translates natural language Javadoc comments into executable specifications.	Preconditions Postconditions Exceptional postconditions	✓
C2S (Zhai et al., 2020)	Code with natural language comments and tests	Translate natural language into JML specifications. They use existing test cases to filter out incorrect specifications.	Preconditions Postconditions Exceptional postconditions	–
ChatGPT (OpenAI, 2022)	Code and query describing what to infer	Large language model that infers the most likely specifications	Any requested form of specifications	✓

References

Ahrendt, Wolfgang, Kovács, Laura, Robillard, Simon, 2015. Reasoning about loops using Vampire in KeY. In: Davis, Martin, Fehnker, Ansgar, McIver, Annabelle, Voronkov, Andrei (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 434–443.

Alsaeed, Ziyad, Young, Michal, 2018. Extending existing inference tools to mine dynamic APIs. In: 2nd IEEE/ACM International Workshop on API Usage and Evolution. WAPI@ICSE 2018, Gothenburg, Sweden, June 2, 2018, ACM, pp. 23–26, Code available at: https://github.com/zalsaeed/enhanceddaikon.

Alshnakat, Anoud, Gurov, Dilian, Lidström, Christian, Rümmer, Philipp, 2020. Constraint-based contract inference for deductive verification. In: Ahrendt, Wolfgang, Beckert, Bernhard, Bubel, Richard, Hähnle, Reiner, Ulbrich, Mattias (Eds.), Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY. In: Lecture Notes in Computer Science, vol. 12345, Springer, pp. 149–176.

Armborst, Lukas, Lathouwers, Sophie, Huisman, Marieke, 2024. Joining forces! reusing contracts for deductive verifiers through automatic translation. In: Herber, Paula, Wijs, Anton (Eds.), Integrated Formal Methods - 18th International Conference, IFM 2023, Leiden, the Netherlands, November 13-15, 2023, Proceedings. In: Lecture Notes in Computer Science, vol. 14300, Springer, pp. 1–19.

Astrauskas, Vytautas, Müller, Peter, Poli, Federico, Summers, Alexander J., 2019. Leveraging Rust types for modular specification and verification. Proc. ACM Program. Lang. 3 (OOPSLA).

Barnett, Mike, Chang, Bor-Yuh Evan, DeLine, Robert, Jacobs, Bart, Leino, K. Rustan M., 2006. Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, Frank S., Bonsangue, Marcello M., Graf, Susanne, de Roever, Willem-Paul (Eds.), Formal Methods for Components and Objects. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 364–387.

Baumann, Christoph, Beckert, Bernhard, Blasum, Holger, Bormer, Thorsten, 2012. Lessons learned from microkernel verification – specification is the new bottleneck. In: Cassez, Franck, Huuck, Ralf, Klein, Gerwin, Schlich, Bastian (Eds.), Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012. In: EPTCS, vol. 102, pp. 18–32.

Beckert, Bernhard, Hähnle, Reiner, 2014. Reasoning and verification: State of the art and current trends. IEEE Intell. Syst. 29 (1), 20–29.

Beckert, Bernhard, Kirsten, Michael, Klamroth, Jonas, Ulbrich, Mattias, 2020. Modular verification of JML contracts using bounded model checking. In: Margaria, Tiziana, Steffen, Bernhard (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. In: Lecture Notes in Computer Science, vol. 12476, Springer, pp. 60–80.

Betts, Adam, Chong, Nathan, Donaldson, Alastair, Qadeer, Shaz, Thomson, Paul, 2012. GPUVerify: A verifier for GPU kernels. SIGPLAN Not. 47 (10), 113–132.

Bierhoff, Kevin, Aldrich, Jonathan, 2008. PLURAL: Checking protocol compliance under aliasing. In: Companion of the 30th International Conference on Software Engineering. In: ICSE Companion '08, Association for Computing Machinery, New York, NY, USA, pp. 971–972.

Blasi, Arianna, Goffi, Alberto, Kuznetsov, Konstantin, Gorla, Alessandra, Ernst, Michael D., Pezzè, Mauro, Castellanos, Sergio Delgado, 2018. Translating code comments to procedure specifications. In: Tip, Frank, Bodden, Eric (Eds.), Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2018, Amsterdam, the Netherlands, July 16-21, 2018, ACM, pp. 242–253, Code available at: https://github.com/albertogoffi/toradocu.

Blom, Stefan, Darabi, Saeed, Huisman, Marieke, Oortwijn, Wytse, 2017. The VerCors tool set: Verification of parallel and concurrent software. In: Polikarpova, Nadia, Schneider, Steve A. (Eds.), Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings. In: Lecture Notes in Computer Science, vol. 10510, Springer, pp. 102–110.

Boockmann, Jan H., Lüttgen, Gerald, 2020. Learning data structure shapes from memory graphs. In: LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. Alicante, Spain, May 22-27, 2020, In: EPiC Series in Computing, vol. 73, EasyChair, pp. 151–168.

Boockmann, Jan H., Lüttgen, Gerald, Mühlberg, Jan Tobias, 2018. Generating inductive shape predicates for runtime checking and formal verification. In: Margaria, Tiziana, Steffen, Bernhard (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Verification. Springer International Publishing, Cham, pp. 64–74.

Boshernitsan, Marat, Doong, Roong-Ko, Savoia, Alberto, 2006. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In: Pollock, Lori L., Pezzè, Mauro (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2006, Portland, Maine, USA, July 17-20, 2006, ACM, pp. 169–180.

Calcagno, Cristiano, Distefano, Dino, O'Hearn, Peter W., Yang, Hongseok, 2011. Compositional shape analysis by means of bi-abduction. J. ACM 58 (6).

Cataño, Néstor, Wahls, Tim, Rueda, Camilo, Rivera, Víctor, Yu, Danni, 2012. Translating b machines to JML specifications. In: Ossowski, Sascha, Lecca, Paola (Eds.), Proceedings of the ACM Symposium on Applied Computing. SAC 2012, Riva, Trento, Italy, March 26-30, 2012, ACM, pp. 1271–1277.

Chalin, Patrice, Kiniry, Joseph R., Leavens, Gary T., Poll, Erik, 2006. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, Frank S., Bonsangue, Marcello M., Graf, Susanne, de Roever, Willem-Paul (Eds.), Formal Methods for Components and Objects. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 342–363.

Cok, David R., 2014. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Dubois, Catherine, Giannakopoulou, Dimitra, Méry, Dominique (Eds.), Proceedings 1st Workshop on Formal Integrated Development Environment. F-IDE 2014, Grenoble, France, April 6, 2014, In: EPTCS, vol. 149, pp. 79–92.

Csallner, Christoph, Tillmann, Nikolai, Smaragdakis, Yannis, 2008. DySy: Dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering. ICSE '08, Association for Computing Machinery, New York, NY, USA, pp. 281–290.

de Oliveira, Steven, Bensalem, Saddek, Prevosto, Virgile, 2016. Polynomial invariants by linear algebra. In: Artho, Cyrille, Legay, Axel, Peled, Doron (Eds.), Automated Technology for Verification and Analysis. Springer International Publishing, Cham, pp. 479–494.

Degiovanni, Renzo, Molina, Facundo, Nolasco, Agustin, Aguirre, Nazareno, Papadakis, Mike, 2023. Specification inference for evolving systems, CoRR abs/2301.12403.

Dohrau, Jérôme, Summers, Alexander J., Urban, Caterina, Münger, Severin, Müller, Peter, 2018. Permission inference for array programs. In: Chockler, Hana, Weissenbacher, Georg (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 55–74.

Ernst, Gidon, 2022. Loop verification with invariants and contracts. In: Finkbeiner, Bernd, Wies, Thomas (Eds.), Verification, Model Checking, and Abstract Interpretation. Springer International Publishing, Cham, pp. 69–92.

Ernst, Michael D., Perkins, Jeff H., Guo, Philip J., McCamant, Stephen, Pacheco, Carlos, Tschantz, Matthew S., Xiao, Chen, 2007. The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69 (1–3), 35–45, Code available at: https://plse.cs.washington.edu/daikon/.

Fiala, Jonáš, Itzhaky, Schachar, Müller, Peter, Polikarpova, Nadia, Sergey, Ilya, 2023. Leveraging Rust types for program synthesis. Proc. ACM Program. Lang. (PLDI).

Filliâtre, Jean-Christophe, Marché, Claude, 2007. The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. In: Lecture Notes in Computer Science, vol. 4590, Springer, pp. 173–177.

Flanagan, Cormac, Leino, K. Rustan M., 2001. Houdini, an annotation assistant for ESC/Java. In: Oliveira, José Nuno, Zave, Pamela (Eds.), FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings. In: Lecture Notes in Computer Science, vol. 2021, Springer, pp. 500–517.

Fraser, Gordon, Arcuri, Andrea, 2011. Evolutionary generation of whole test suites. In: Núñez, Manuel, Hierons, Robert M., Merayo, Mercedes G. (Eds.), Proceedings of the 11th International Conference on Quality Software. QSIC 2011, Madrid, Spain, July 13-14, 2011, IEEE Computer Society, pp. 31–40.

Furia, Carlo A., Meyer, Bertrand, Velder, Sergey, 2014. Loop invariants: Analysis, classification, and examples. 46 (3).

Galeotti, Juan Pablo, Furia, Carlo A., May, Eva, Fraser, Gordon, Zeller, Andreas, 2014. DynaMate: Dynamically inferring loop invariants for automatic full functional verification. In: Yahav, Eran (Ed.), Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. In: Lecture Notes in Computer Science, vol. 8855, Springer, pp. 48–53, Code available at: https://www.st.cs.uni-saarland.de/dynamate/.

Garg, Aayush, Degiovanni, Renzo, Molina, Facundo, Papadakis, Mike, Aguirre, Nazareno, Cordy, Maxime, Traon, Yves Le, 2023. Assertion inferring mutants, CoRR.

Giorgetti, Alain, Groslambert, Julien, 2006. JAG: JML annotation generation for verifying temporal properties. In: Baresi, Luciano, Heckel, Reiko (Eds.), Fundamental Approaches To Software Engineering, 9th International Conference, FASE 2006, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings. In: Lecture Notes in Computer Science, vol. 3922, Springer, pp. 373–376.

Grunwald, Daniel, Gladisch, Christoph, Liu, Tianhai, Taghdiri, Mana, Tyszberowicz, Shmuel S., 2014. Generating JML specifications from alloy expressions. In: Yahav, Eran (Ed.), Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. In: Lecture Notes in Computer Science, vol. 8855, Springer, pp. 99–115.

Gurov, Dilian, Lidström, Christian, Nyberg, Mattias, Westman, Jonas, 2017. Deductive functional verification of safety-critical embedded C-code: An experience report. In: Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings. In: Lecture Notes in Computer Science, vol. 10471, Springer, pp. 3–18.

Hähnle, Reiner, Huisman, Marieke, 2019. Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science - State of the Art and Perspectives. In: Lecture Notes in Computer Science, vol. 10000, Springer, pp. 345–373.

Hamie, Ali, 2004. Translating the object constraint language into the java modelling language. In: Proceedings of the 2004 ACM Symposium on Applied Computing. SAC '04, Association for Computing Machinery, New York, NY, USA, pp. 1531–1535.

Hangal, Sudheendra, Lam, Monica S., 2002. Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02, Association for Computing Machinery, New York, NY, USA, pp. 291–301.

Huisman, Marieke, Monti, Raúl E., 2020. On the industrial application of critical software verification with VerCors. In: Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III. In: Lecture Notes in Computer Science, vol. 12478, Springer, pp. 273–292.

Huisman, Marieke, Tamalet, Alejandro, 2009. A formal connection between security automata and JML annotations. In: Chechik, Marsha, Wirsing, Martin (Eds.), Fundamental Approaches To Software Engineering, 12th International Conference, FASE 2009, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. In: Lecture Notes in Computer Science, vol. 5503, Springer, pp. 340–354.

Humenberger, Andreas, Jaroschek, Maximilian, Kovács, Laura, 2018. Aligator.jl – A Julia package for loop invariant generation. In: Rabe, Florian, Farmer, William M., Passmore, Grant O., Youssef, Abdou (Eds.), Intelligent Computer Mathematics. Springer International Publishing, Cham, pp. 111–117.

Jacobs, Bart, Smans, Jan, Philippaerts, Pieter, Vogels, Frédéric, Penninckx, Willem, Piessens, Frank, 2011. VeriFast: A powerful, sound, predictable, fast verifier for c and Java. In: Bobaru, Mihaela Gheorghiu, Havelund, Klaus, Holzmann, Gerard J., Joshi, Rajeev (Eds.), NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. In: Lecture Notes in Computer Science, vol. 6617, Springer, pp. 41–55.

Johansson, Moa, 2019. Lemma discovery for induction. In: Kaliszyk, Cezary, Brady, Edwin, Kohlhase, Andrea, Sacerdoti Coen, Claudio (Eds.), Intelligent Computer Mathematics. Springer International Publishing, Cham, pp. 125–139.

Knüppel, Alexander, Thüm, Thomas, Pardylla, Carsten, Schaefer, Ina, 2018. Experience report on formally verifying parts of OpenJDK's API with KeY. In: Proceedings 4th Workshop on Formal Integrated Development Environment. F-IDE@FLoC 2018, Oxford, England, 14 July 2018, In: EPTCS, vol. 284, pp. 53–70.

Lathouwers, Sophie, Huisman, Marieke, 2022. Formal specifications investigated: A classification and analysis of annotations for deductive verifiers. In: Gnesi, Stefania, Plat, Nico (Eds.), 10th IEEE/ACM International Conference on Formal Methods in Software Engineering. FormaliSE@ICSE 2022, Pittsburgh, PA, USA, May 22-23, 2022, IEEE.

Le, Quang Loc, Gherghina, Cristian, Qin, Shengchao, Chin, Wei-Ngan, 2014. Shape analysis via second-order bi-abduction. In: Biere, Armin, Bloem, Roderick (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 52–68.

Le, Ton Chanh, Zheng, Guolong, Nguyen, ThanhVu, 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In: McKinley, Kathryn S., Fisher, Kathleen (Eds.), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, ACM, pp. 788–801, Code available at: https://github.com/guolong-zheng/sling/.

Leavens, Gary T., Baker, Albert L., Ruby, Clyde, 2006. Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes 31 (3), 1–38.

Leavens, Gary T, Poll, Erik, Clifton, Curtis, Cheon, Yoonsik, Ruby, Clyde, Cok, David, Müller, Peter, Kiniry, Joseph, Chalin, Patrice, Zimmerman, Daniel M, et al., 2008. JML reference manual.

Luo, Chenguang, Craciun, Florin, Qin, Shengchao, He, Guanhua, Chin, Wei-Ngan, 2010. Verifying pointer safety for programs with unknown calls. J. Symbolic Comput. 45 (11), 1163–1183, Special Issue on Invariant Generation and Advanced Techniques for Reasoning about Loops.

Meyer, Bertrand, 2002. Design by Contract. Prentice Hall Upper Saddle River.

Mohsen, Mahmoud, Jacobs, Bart, 2016. One step towards automatic inference of formal specifications using Automated VeriFast. In: ter Beek, Maurice H., Gnesi, Stefania, Knapp, Alexander (Eds.), Critical Systems: Formal Methods and Automated Verification - Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings. In: Lecture Notes in Computer Science, vol. 9933, Springer, pp. 56–64.

Molina, Facundo, d'Amorim, Marcelo, Aguirre, Nazareno, 2022. Fuzzing class specifications. In: Proceedings of the 44th International Conference on Software Engineering. ICSE '22, Association for Computing Machinery, New York, NY, USA, pp. 1008–1020.

Molina, Facundo, Ponzio, Pablo, Aguirre, Nazareno, Frias, Marcelo F., 2021. EvoSpex: An evolutionary algorithm for learning postconditions. In: 43rd IEEE/ACM International Conference on Software Engineering. ICSE 2021, Madrid, Spain, 22-30 May 2021, IEEE, pp. 1223–1235, Available at: https://zenodo.org/record/4458256#.YAyzdpMzagQ.

Müller, Peter, Schwerhoff, Malte, Summers, Alexander J., 2016. Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, Barbara, Leino, K. Rustan M. (Eds.), Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings. In: Lecture Notes in Computer Science, vol. 9583, Springer, pp. 41–62.

Nguyen, ThanhVu, Dwyer, Matthew B., Visser, Willem, 2017. SymInfer: inferring program invariants using symbolic states. In: Rosu, Grigore, Penta, Massimiliano Di, Nguyen, Tien N. (Eds.), Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, IEEE Computer Society, pp. 804–814.

Nguyen, Hoan Anh, Dyer, Robert, Nguyen, Tien N., Rajan, Hridesh, 2014a. Mining preconditions of APIs in large-scale code corpus. In: Cheung, Shing-Chi, Orso, Alessandro, Storey, Margaret-Anne D. (Eds.), Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE-22, Hong Kong, China, November 16 - 22, 2014, ACM, pp. 166–177.

Nguyen, ThanhVu, Kapur, Deepak, Weimer, Westley, Forrest, Stephanie, 2014b. DIG: A dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol. 23 (4), 30:1–30:30, Code available at: https://github.com/dynaroars/dig.

Nimmer, Jeremy W., Ernst, Michael D., 2002. Invariant inference for static checking: An empirical evaluation. SIGSOFT Softw. Eng. Notes 27 (6), 11–20.

OpenAI, 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt#OpenAI. Tool: https://chat.openai.com. (Accessed 15 May 2023).

Padhi, Saswat, Sharma, Rahul, Millstein, Todd, 2016. Data-driven precondition inference with learned features. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '16, Association for Computing Machinery, New York, NY, USA, pp. 42–56.

Pham, Long H., Thi, Lyly Tran, Sun, Jun, 2017. Assertion generation through active learning. In: Uchitel, Sebastián, Orso, Alessandro, Robillard, Martin P. (Eds.), Proceedings of the 39th International Conference on Software Engineering. ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume, IEEE Computer Society, pp. 155–157, Code available at: https://github.com/sunjun-group/Ziyuan.

Polikarpova, Nadia, Ciupa, Ilinca, Meyer, Bertrand, 2009. A comparative study of programmer-written and automatically inferred contracts. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ISSTA '09, Association for Computing Machinery, New York, NY, USA, pp. 93–104.

Raghavan, Arun, Leavens, Gary, 2005. Desugaring JML method specifications. Comput. Sci. Tech. Rep. 345.

Richter, Cedric, Wehrheim, Heike, 2019. PeSCo: Predicting sequential combinations of verifiers. In: Beyer, Dirk, Huisman, Marieke, Kordon, Fabrice, Steffen, Bernhard (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, pp. 229–233.

Sadiq, Ayesha, Li, Li, Li, Yuan-Fang, Ahmed, Ijaz, Ling, Sea, 2019. Sip4J: Statically inferring access permission contracts for parallelising sequential Java programs. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, pp. 1098–1101, Code available at: https://github.com/AyeshaSadiq7/Sip4J.

Scheben, Christoph, 2014. Program-level Specification and Deductive Verification of Security Properties (Ph.D. thesis). Karlsruhe Institute of Technology.

Scott, Joseph, Niemetz, Aina, Preiner, Mathias, Nejati, Saeed, Ganesh, Vijay, 2021. MachSMT: A machine learning-based algorithm selector for SMT solvers. In: Groote, Jan Friso, Larsen, Kim Guldstrand (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer International Publishing, Cham, pp. 303–325.

Si, Xujie, Dai, Hanjun, Raghothaman, Mukund, Naik, Mayur, Song, Le, 2018. Learning loop invariants for program verification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (Eds.), In: Advances in Neural Information Processing Systems, vol. 31, Curran Associates, Inc..

Singher, Eytan, Itzhaky, Shachar, 2021. Theory exploration powered by deductive synthesis. In: Silva, Alexandra, Leino, K. Rustan M. (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 125–148.

Singleton, John L., Leavens, Gary T., Rajan, Hridesh, Cok, David R., 2018. An algorithm and tool to infer practical postconditions. In: Chaudron, Michel, Crnkovic, Ivica, Chechik, Marsha, Harman, Mark (Eds.), Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp. 313–314.

Srivastava, Saurabh, Gulwani, Sumit, 2009. Program verification using templates over predicate abstraction. SIGPLAN Not. 44 (6), 223–234.

Tabar, Asmae Heydari, Bubel, Richard, Hähnle, Reiner, 2022. Automatic loop invariant generation for data dependence analysis. In: Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering. FormaliSE '22, Association for Computing Machinery, New York, NY, USA, pp. 34–45.

Tran-Jørgensen, Peter W.V., Larsen, Peter Gorm, Leavens, Gary T., 2018. Automated translation of VDM to JML-annotated Java. Int. J. Softw. Tools Technol. Transf. 20 (2), 211–235.

Vogels, Frédéric, Jacobs, Bart, Piessens, Frank, Smans, Jan, 2011. Annotation inference for separation logic based verifiers. In: Bruni, Roberto, Dingel, Jürgen (Eds.), Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings. In: Lecture Notes in Computer Science, vol. 6722, Springer, pp. 319–333.

Wasser, Nathan Daniel, 2017. Automatic Generation of Specifications using Verification Tools (Ph.D. thesis). Technische Universit.

Weiß, Benjamin, 2009. Predicate abstraction in a program logic calculus. In: Leuschel, Michael, Wehrheim, Heike (Eds.), Integrated Formal Methods, 7th International Conference, IFM 2009, DÜsseldorf, Germany, February 16-19, 2009. Proceedings. In: Lecture Notes in Computer Science, vol. 5423, Springer, pp. 136–150.

Zhai, Juan, Shi, Yu, Pan, Minxue, Zhou, Guian, Liu, Yongxiang, Fang, Chunrong, Ma, Shiqing, Tan, Lin, Zhang, Xiangyu, 2020. C2S: translating natural language comments to formal program specifications. In: Devanbu, Prem, Cohen, Myra B., Zimmermann, Thomas (Eds.), ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event, USA, November 8-13, 2020, ACM, pp. 25–37.

Sophie Lathouwers recently received a Ph.D. degree in Computer Science at the University of Twente in the Netherlands. Her research interests include specifications, software verification, software engineering, tool usability, taxonomies and open research data. She was one of the co-organizers of the first Alice & Eve workshop.

Marieke Huisman is a professor in Software Reliability at the University of Twente. She is well-known for her work on program verification of concurrent software. In 2011, she obtained an ERC Starting Grant, with which she started development of the VerCors verifier, a tool for the verification of concurrent software. Since 2017, as part of her NWO personal VICI grant Mercedes, she further improved verification support, by enabling the verification of a larger class of properties, and by making verification more automatic. She was awarded the Netherlands Prize for ICT Research in 2013 and the Athena Award in 2023.