Locking Bugs Out with KeY

A Case Study on Automated Formal Verification of Java Programs

Tejas Kochar[1]
Supervisor(s): Benedikt Ahrens[1]**, Kobe Wullaert**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
22nd June 2025

Name of the student: Tejas Kochar
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Kobe Wullaert, Maliheh Izadi

An electronic version of this thesis is available at https://repository.tudelft.nl/.
Code for this thesis can be found at https://github.com/Tejas-Kochar/key-verifier/.

Abstract

KeY prover has been used to verify parts of the OpenJDK library and Norwegian election software, making it one of the most capable tools for formally verifying Java programs. However, an intuitive explanation of its theoretical foundations, capabilities and limitations is not available and the tool itself has a steep learning curve. This greatly increases the amount of effort to understand the tool sufficiently well to evaluate its suitability for ones needs. Here we implement two case studies to develop an intuition for the working of the tool in practice, exploring what properties can or cannot be expressed as specifications and what verifying correctness with KeY means. We find that the learning curve for KeY is even steeper than expected and conclude that it can be used to verify very complex functional and even some non-functional properties of sequential Java 7 code, conditioned on the capabilities of the user.

Keywords: Automated Formal Verification, KeY, Deductive Verification, Java

Acknowledgements

I would like to express my gratitude to my Supervisor Kobe Wullaert and Professor Dr. Benedikt Ahrens for their invaluable guidance throughout this study. Their feedback was instrumental in shaping this research. I would also like to thank my peers, Mohammed Balfakeih, Dinu Blanovschi, Jeroen Koelewijn and Kajetan Neumann for their support and many ideas and suggestions throughout the course of this study.

1 Introduction

Most would agree that it is good practice to test software before putting it into production. However, in critical systems, where failure of software can be disastrous, a higher level of correctness assurance is desired and often required by law. Formal Verification is a method used to provide such assurances about software by mathematically proving its correctness.

As the size and complexity of code increases, verification tools become essential to (partially) automate and simplify the process. KeY [1] is one such tool, designed to semi-automatically prove the correctness of sequential Java code through deductive verification.

Recent works like the verification of Norwegian election software [2] and parts of the OpenJDK library [3]–[8] demonstrate the capability of KeY Prover, with bugs being found in classes like LinkedList and OpenJDKs default sorting algorithm that are relied on by applications running on billions of devices. Other literature on KeY includes extensions to support more features of Java [9]–[12], tutorials to get members of the formal verification community started with KeY [13]–[15], and a textbook [1]. While these works push the boundaries of what can be done with KeY or provide guidance on using it, answers to questions about what verified correctness means, the kind of guarantees KeY can provide, or the assumptions it depends on are seemingly considered implicit and left unanswered. Answering these questions is necessary for anyone that wishes to avoid unpleasant surprises where programs verified to be 'correct' behave in unexpected ways.

To make this knowledge easily available and understandable, we examined relevant literature like the textbook [1] to understand the logical foundations, implemented two case studies – a sorting algorithm and a key store data structure – and then gave our answers to the questions based on our experience and reflected on the capabilities and limitations of the KeY Prover.

In doing this, we found that the learning curve was steeper than we expected due to the interactive nature of the prover requiring knowledge and experience to recognize where the prover gets stuck in attempted proofs and what guidance is required to finish the proofs. Limited documentation and support exacerbated this problem. While the tool is hard to use for beginners, its support for interaction makes it capable of verifying complex real-world software that often cannot be refactored to be made suitable for verification.

2 Background

This section explains how the KeY prover works in three steps. Section 2.1 explains how specifications can be provided as annotations in regular Java programs. Section 2.2 motivates why KeY uses Dynamic Logic for internally representing the input. Finally, Section 2.3 explains how KeY uses a suitable calculus, Sequent Calculus in its attempts to construct proofs of 'correctness'.

2.1 What does the KeY Prover take as input?

The KeY prover is designed to formally verify that a program always respects a formal specification of its expected behaviour. In other words, it takes a formal specification and a program as input and attempts to prove that the two match. An example of an input to KeY to verify is given in Listing 1 below. The example is taken from the KeY textbook [1, Chapter 16] .

The listing contains a regular Java class `PostInc`, which is the program to verify, and some comments containing Java-like statements which are the specifications. `PostInc` has two `int` fields `x` and `y` declared in line 3 and a field `rec` of type `PostInc` in line 2. It has only one method, `postInc()` in lines 11-13, that updates the fields `x` and `y` of `rec` by setting `x` to `y`, and then incrementing `y`.

The specifications given in comments are in a modeling language called Java Modeling Language (JML), and are required to start with the `@` symbol. The comments from line 7-10 are a **contract** for the method `postInc()`. Contracts consist of the **pre-conditions** and **post-conditions** that must hold for a method, specified through the `ensures` and `requires` keywords, respectfully. We note that pre-conditions refer to variables in the state before the execution of the method and post-conditions refer to the state after execution. The `\old(...)` expressions in line 9 are used to refer to variables in their state before method execution in post-conditions. This enables making assertions about change in state. For instance, the expression `rec.y == \old(rec.y)+1` ensures that `rec.y` gets incremented by one by calling the method.

The word `public` on line 7 is an **access specifier** setting the visibility of the contract from other classes, and `normal_behavior` is short for stating that *if the pre-condition holds, the program terminates and does so without raising any exceptions.*

The comment on line 5, starting with `invariant` is called a **class invariant**, which is an expression that must hold before and after the execution of all methods.

Listing 1: Example of program with specifications taken as input by KeY.

```
1  public class PostInc {
2      public PostInc rec;
3      public int x,y;
4
5      //@ public invariant rec.x >= 0 && rec.y>= 0;
6
7      /*@ public normal_behavior
8        @ requires true;
9        @ ensures rec.x == \old(rec.y) && rec.y == \old(rec.y)+1;
10       @*/
11     public void postInc() {
12         rec.x = rec.y++;
13     }
14 }
```

2.2 What System of Logic does KeY use?

From the above listing, we see that 'formal specifications' for KeY are composed of Java-like expressions that evaluate to a boolean value. These statements could be expressed as predicates, i.e. in First Order Logic (FOL). However, composing these statements requires something more than FOL as it has no constructs for expressing that a predicate should hold *before* or *after* the execution of a program. In other words, it has no notion of state.

To address this gap in expressiveness, KeY uses Dynamic Logic (DL), which extends FOL with a construct called **modality** to express state and talk about programs and assertions in the same language. Modalities are denoted as either `<p>` or `[p]`, and mean the *final state* of `p`, where `p` is a valid program. Modalities are followed by assertions, with the expression `<p>q` meaning that the program `p` ends in a state in which `q` holds. In other words, `q` is a post-condition for `p`.

Thus, the contract for a method M stating that *if the pre-condition `P` holds for the state just before calling `M`, then `M` is required to always terminate in a state in which post-condition `Q` holds*, can be expressed in DL as `P` $\rightarrow$ `<M>Q`. From this, we can see that formally verifying whether `M` adheres to it's contract or not can be reduced to showing that `P` $\rightarrow$ `<M>Q` is a tautology (a statement that is always true). KeY takes the approach of deductive verification, i.e. logical inference, to achieve this.

2.3 How does KeY perform deductive verification?

The following explanation of deductive verification up to the weakest pre-condition computation example follows a history of deductive verification given by Reiner Hähnle and Marieke Huisman [16].

Deductive Verification has its roots in Floyd-Hoare logic and Dijkstra's weakest preconditions. Hoare built on Floyd's work to introduce the Hoare triple, `{P}S{Q}`, stating that if a program `S` is executed in a state in which `P` holds, then `Q` will hold for the state after the termination of the program, provided it terminates [17], [18]. We can recognise this as a formulation of the modalities we just saw for dynamic logic, i.e. `P` $\rightarrow$ `<S>Q`, and indeed, the original version of dynamic logic was introduced by Pratt [19] based on Floyd-Hoare logic.

Floyd and Hoare proposed a set of syntactic proof rules to prove the correctness of an algorithm. For instance, if we suppose the program `S` is composed of two statements $S_1; S_2$, and we wish to show that `{P}S{Q}` holds, then we can apply the proof rule for statement composition to simplify the task:

$$\frac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

This big-step semantics [20] rule expresses that to prove that if $S_1; S_2$ is executed in a state satisfying precondition `P`, and if it terminates in a state satisfying `Q`, it is sufficient to find an intermediate assertion `R`, such that `R` can be established as a post-condition for the first statement S_1, and is a sufficient pre-condition for S_2 to establish post-condition `Q`. What we see here is that the correctness problem for a complete algorithm `S` can be broken up into a correctness problem of its individual instructions S_1 and S_2.

Dijkstra observed that it is possible to compute the minimal pre-condition that is necessary to guarantee that a program will establish a given post-condition [21]. This simplifies verification, because one no longer has to 'invent' the intermediate predicate that describes the state between two statements, but this can be computed. In particular, the weakest precondition `wp` for a statement $S_1; S_2$ can be computed using the following rule:

$$\texttt{wp}(S_1; S_2, Q) = \texttt{wp}(S_1, \texttt{wp}(S_2, Q))$$

More rules exist for other kinds of instructions. These rules can be applied repeatedly by matching the expression being proven with the expression below the horizontal line of proof rules, and applying them upwards by substituting the matching part of the expression with the part of the proof rule above the horizontal line. This process results in a proof tree that grows upwards, possibly spreading into multiple branches (as for example with the rule of composition given above), and with the expressions ideally getting progressively simpler until they can be reduced to `true`. A proof is complete when all branches can be reduced to `true`.

The question of what does proving correctness with KeY mean becomes a question of what can be specified in a contract, because KeY simply ensures that the program being verified adheres to it's contract. Answering this requires looking into its expressive capabilities and also it's default assumptions, which we do by the means of case studies in Section 4. The question is finally answered in Section 5.1.

3 Problem Description

We perform two case studies in verifying software with KeY – one on a sorting algorithm and the other on a key store data structure. Correctness criteria for verification in the two case studies are presented in Section 3.1 and Section 3.2 respectively.

3.1 Sorting algorithm

A sorting algorithm takes as input a sequence of elements belonging to a totally ordered set and produces a permutation of the input sequence which respects the total ordering. That is, the following three properties must hold for the sorting algorithm –

Termination:	the algorithm terminates in finite time for any finite input
Permutation:	the output is a permutation of the input
Monotonicity:	the output elements are in non-decreasing order

3.2 Key Store

A key store is a data structure that maintains an association from a set of keys to a collection of values. It minimally supports operations like creating or deleting associations and using a key to get the associated value if a mapping exists. The operations can be specified as –

`get(int key):`	Returns `value` associated with `key`, or `null` if no mapping for `key` is found
`put(int key, Object value):`	Ensures that calling `get(key)` returns `value`
`remove(int key):`	Ensures that calling `get(key)` returns `null`

Some additional constraints placed on the store are -

1. there cannot be two mappings from the same key
2. the store must not be modified in any way other than specified above
3. all methods must terminate in finite time on all inputs without raising an exception.

4 Case Studies

Our approach to the case studies was to learn the basics of using KeY by following the tutorials in the textbook [1, Part IV] and then starting with the case studies themselves, as opposed to first studying the textbook thoroughly and only then starting with the case studies. This was done considering the fixed time of less than 10 weeks available to complete the study, and because the case studies themselves appeared to be quite simple. The intention was to quickly finish the cases, learning whatever is required in the process and then expand the scope of the project. However, we only managed to completely prove the correctness of a simple key store and partially prove the correctness of the insertion sort sorting algorithm.

We initially planned on verifying Selection Sort but found that the tutorial verified it as an example [14]. Since the example looked very straightforward, we decided to start doing the key store first and return later to do Insertion Sort. However, we first present the implementation and specification details for the sorting algorithm case study in Section 4.1 and then the key store case study in Section 4.2.

4.1 Sorting Algorithm

This section covers the implementation details of the **Insertion Sort** algorithm in Section 4.1.1 and the specification details in Section 4.1.2. Section 4.1.3 briefly explains how we arrived at the implementation we used. We also make use of this case study to explain some more constructs like loop invariants used for writing specifications.

4.1.1 Implementation of Insertion Sort

We implemented a modified version of the insertion sort algorithm, shown in Listing 2. The implementation consists of a class `SwapInsertionSort` with the sorting method `void sort(int[] a)` in lines 2-6 which sorts an `int[]` array `a` in-place in non-decreasing order. It does this using two helper methods: `void shift(int[] a, int i)` in lines 7-10, which shifts the `i`'th element of `a` to its correct relative position in the sub-array `a[0..i]`, and `void swap(int a[], int j, int j1)` in lines 12-16, which swaps the elements of `a` at indices `j` and `j1`.

The algorithm conceptually divides its input array into a sorted initial segment comprising of the first element `a[0]` of the array and the remaining unsorted segment. The variable `i` introduced in the `for` loop in line 3 denotes the boundary between the two segments and itself lies in the unsorted part. With each iteration of the loop, the algorithm shifts the element just beyond the boundary, at the `i`'th index, to its correct relative position in the sorted segment of the array by calling `shift` in line 4, thus growing the sorted segment (indicated by `i` being incremented in line 3) until the entire array is sorted. This implementation of insertion sort deviates from typical implementations by using swaps to perform the shifting.

Listing 2: Implementation of the Insertion Sort algorithm

```
1  class SwapInsertionSort {
2      void sort(int[] a) {
3          for (int i = 1; i < a.length; i++) {
4              shift(a, i);
5          }
6      }
7      void shift(int[] a, int i) {
8          for (int j = i-1; j >= 0 && a[j] > a[j+1]; j --) {
9              swap(a, j, j+1);
10         }
11     }
```

```
12      void swap(int[] a, int j, int j1) {
13          int tmp = a[j];
14          a[j] = a[j1];
15          a[j1] = tmp;
16      }
17  }
```

4.1.2 Specification and Verification of Insertion Sort

Listing 3 presents specifications sufficient for KeY to automatically verify the correctness of the `sort` method. The contract for `sort`, given in lines 1, 2 and 3 express the termination, permutation and monotonicity properties from the problem description in Section 3.1 respectively. In particular, the permutation property on line 2 makes use of the in-built Algebraic Data Type (ADT) for sequences, `\seq`, to represent `a` in the states before and after the method and then makes use of an in-built predicate `\dl_seqPerm` to check whether the two sequences are permutations of each other. We also note the use of the quantifier $\forall$ expressed as `\forall` in line 3, to introduce a variable `int x`, and then constrain its value in `[0..i-2]` inclusive and state that `a[x] <= a[x+1]`.

The body of `sort` uses an unbounded loop (i.e. the number of executions is not fixed). Reasoning about unbounded loops in general is difficult. Most verification approaches make use of **loop invariants** to reason about programs containing loops. Loop invariants are conditions that hold in the program state immediately before entering a loop and after each execution of the loop body. This means that the invariant will also hold if the loop termination guard becomes false, making the loop invariant available to use for reasoning after the loop. In the case of `sort`, if we combine the loop guard `i < a.length` with the loop invariant on line 6 stating `0 <= i <= a.length`, we see that the loop can only terminate when `i == a.length`. If we further combine this with the invariant on line 8, which states that montonicity is established for the initial segment `a[0..i]`, we get that the entire array is sorted on termination of the loop. The invariant on line 7 states that `a` is only permuted, thus showing that if the loop terminates, then `a` is 'correctly' sorted. Only two things remain to be shown to prove the correctness of `sort` – that the loop invariants specified actually are loop invariants, and that the loop terminates.

For KeY to prove that the loop terminates, it is sufficient for it to prove that the expression in the `decreases` clause on line 10 strictly decreases after each iteration, and is bounded from below. We can see that these are indeed true as the term `a.length-i` strictly decreases because `i` strictly increases, and is bounded from below because `a.length` is constant and `i` is bounded from above.

KeY proves the validity of loop invariants inductively. This involves showing that they hold before the start of the loop, which can easily be seen in our case, and to show that they hold after an arbitrary execution of the loop given that they held before it. This means showing that the loop body, which in this case only contains a call to `shift`, always preserves the loop invariants. By default, KeY deals with method calls by first proving that the pre-conditions specified in its contract hold and then assuming that the post-conditions hold after the method call. It is straightforward to see that the pre-conditions of `shift` hold because of the loop invariants being true before the method call and that its post-conditions on lines 20 and 21 correspond exactly to the loop invariants on lines 8 and 7. Thus, the prover is able to establish the correctness of `sort`.

Listing 3: Specifications for `sort` and `shift` methods

```
1       /*@ normal_behaviour
2         @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
```

```
3       @ ensures (\forall int x; 0 <= x < a.length-1; a[x] <= a[x+1]);
4       @*/
5     void sort(int[] a) {
6         /*@ loop_invariant 1 <= i <= a.length;
7           @ loop_invariant \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
8           @ loop_invariant (\forall int x; 0 <= x < i-1; a[x] <= a[x+1]);
9           @ assignable a[*];
10          @ decreases a.length - i;
11          @*/
12        for (int i = 1; i < a.length; i++) {
13            shift(a, i);
14        }
15    }
16
17    /*@ normal_behaviour
18      @ requires 1 <= i < a.length;
19      @ requires (\forall int x; 0 <= x < i-1; a[x] <= a[x+1]);
20      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
21      @ ensures (\forall int x; 0 <= x < i; a[x] <= a[x+1]);
22      @ assignable a[0..i+1];
23      @*/
24    void shift(int[] a, int i) {...}
```

While we have proven the 'correctness' of `sort`, this correctness is conditioned on the correctness of the implementation of `shift` used. We were, however, unable to automatically verify the correctness of the implementations of `shift` and `swap`. Their complete implementations and specifications are available in Listing 9 in Appendix A, and a detailed discussion is omitted due to their similarity to the specification for `sort`.

4.1.3 Process

We started with a regular implementation of Insertion Sort and broke it into two methods, `sort` and `shift` to make it similar to how Selection Sort was done using sort and max in the book [1]. However, it was unable to verify that the `shift` method only permuted the array (this was more complicated than the case of Selection Sort as the permutation property is temporarily violated in the loop performing the shifting and gets re-established after). So we changed the implementation to only use swaps to modify the array, like the given examples of Selection Sort and Quick Sort. However, we could not prove it for this implementation either, so we extracted `swap` into a separate method and tried proving that it only permutes an array. This also failed to complete automatically and requires user guidance. We were also unable to establish the monotonicity property of shift, but believe we can get the prover to do this automatically if we spend some more time on this.

4.2 Key-Value store

This sub-section goes over the implementation and specification of a key store for verification in Section 4.2.1 and Section 4.2.2 respectively. Section 4.2.3 explains how we arrived at the implementation we verified in the end.

4.2.1 Implementation

We implemented a simple key store allowing for keys of type `int`, values of type `Object`, and performing get, put and remove operations in linear time. The implementation is given in Listing 4. It omits the bodies of the `get` and `remove` methods as they are similar to the `put` method.

The listing contains the class `KeyStore` with the private fields `int[] keys`, `Object[] values` and `int size` in lines 2-4 for keeping track of the keys, values and number of entries in the data structure respectively. It has the methods `get`, `put` and `remove`, which each rely on the helper method `findIndex` in lines 6-13 to iterate over the store and find the index at which a specific key is stored in the data structure, if at all present. The methods then perform their respective operations on the entry located at the returned index. The implementations of `get` and `remove` along with other methods like `enlarge` can be found in the appendix in Listing 10.

Listing 4: Implementation of `KeyStore`

```
1  public class KeyStore {
2      private int[] keys;
3      private Object[] values;
4      private int size;
5
6      private int findIndex(int key) {
7          for(int i = 0; i < size; i++) {
8              if(keys[i] == key) {
9                  return i;
10             }
11         }
12         return -1;
13     }
14     public Object get(int key) {...}
15     public void put(int key, Object value) {
16         int index = findIndex(key);
17         if (index == -1) {
18             keys[size] = key;
19             values[size++] = value;
20         }
21         else {
22             values[index] = value;
23         }
24     }
25     public void remove(int key) {...}
26 }
```

4.2.2 Specification

Specifications for data structures often include constraints limiting the states that the structure can legally be in, in addition to the conditions on the inputs and outputs of methods operating on the structure. Such constraints on the state of an object, which must be respected once the execution of any method finishes, are called **class invariants**. They can be specified using the keyword `invariant`, which is shorthand for adding the condition as a pre and post condition to the contracts of each method of the class. Listing 5 contains the invariants that the key store operations must maintain. For example, the invariant on line 6 states that keys in the key store must be unique. The other invariants, such as the lengths of the `keys` and `values` arrays being the same, or the `size` of the key store not exceeding the lengths of the underlying arrays, are specific to the implementation.

We note that line 1 of the listing shows the `nullable` annotation being added to the declaration of `values`, the invariant on line 2 states that the array `values` itself cannot be `null` and the invariant on line 5 states that the elements of `values` in use for storing entries in the key store cannot be `null`. Put together, their meaning is that only elements of `values` that are not in use for storing entries can be `null`.

Listing 5: Class invariants for `KeyStore`

```
1     private /*@ nullable @*/ Object[] values;
2     /*@ private invariant keys != null && values != null && keys != values;
3       @ private invariant keys.length == values.length;
4       @ private invariant 0 <= size && size <= keys.length;
5       @ private invariant (\forall int i; 0 <= i && i < size; values[i] != null);
6       @ private invariant (\forall int x, y; 0 <= x && x < y && y < size;
            keys[x] != keys[y]);
7       @ private invariant \typeof(keys) == \type(int[]) && \typeof(values) ==
            \type(Object[]);
8       @*/
```

Listing 6 shows a **model method** `boolean contains(int key)` that is used in the specifications of most other methods to express the presence or absence of a particular key. The restrictions placed on model methods are that they are only available for use in specifications, they must be `pure` and that their body, if provided, must always terminate. However, they also significantly increase the expressiveness of JML as model methods can make use of JML expressions in the body, or be made abstract (defining the method signature but not the body) and instead rely on pre and post conditions to declare their behaviour. Such 'abstract' declarations of methods can be used in specifications because the methods are never run on concrete inputs and their contracts suffice to reason about them. In our case, we haven't defined a contract for `contains` but instead use it as shorthand for the predicate returned by its body.

Listing 6: Definition of the model method `contains`

```
1     /*@ model public boolean contains(int key) {
2       @     return (\exists int i; 0 <= i && i < size; keys[i] == key);
3       @ }
4       @*/
```

The specification for `put` in Listing 7 below is actually composed of two contracts, separated by the keyword `also` on line 6. The first contract is for the case when the store does not contain a mapping for key, and ensures that calling `get(key)` just after `put` returns `value`, the variable `size` is incremented and that all other entries are unmodified. The method also expects `size` to be strictly less than `keys.length`, indicating that there is space in the store for another entry. This is a design decision which puts the responsibility of ensuring sufficient space in the data structure on the client. We note that because the method `get` is `pure` (without side effects), we are able to use it in the specification.

The second contract is quite comparable to the first, only being for the complementary case of a mapping for `key` already being present in the key store. It ensures that the old value associated to `key` is replaced with the new one, calling `get` with `key` returns `value`, and `size` and other entries are not changed.

Listing 7: Specification of the `put` method

```
1     /*@ public normal_behaviour
2       @ requires !contains(key) && size < keys.length;
3       @ ensures get(key) == value &&  size == \old(size) + 1;
4       @ ensures (\forall int x; 0 <= x < size - 1; keys[x] == \old(keys[x]) &&
            values[x] == \old(values[x]));
5       @ assignable keys[*], values[*], size;
6       @ also
7       @ public normal_behaviour
8       @ requires contains(key);
9       @ ensures get(key) == value && size == \old(size);
10      @ ensures (\forall int x; 0 <= x < size; keys[x] == \old(keys[x]) && (key
            ==  keys[x] || values[x] == \old(values[x])));
```

```
11        @ assignable values[*];
12        @*/
13      public void put(int key, Object value) {
14          int index = findIndex(key);
15          if (index == -1) {
16              keys[size] = key;
17              values[size++] = value;
18          }
19          else {
20              values[index] = value;
21          }
22      }
```

Since the `put` method makes use of `findIndex`, its correctness depends on the implementation of `findIndex` also adhering to its specification, given in Listing 8. The contract in lines 1-4 simply states that the method returns `-1` if the key store has no entry for the parameter `key` and returns its index otherwise. The `pure` annotation bars side effects and the loop invariant in the method body states that the key being searched for is not present in the sub-array `keys[0..i]`. This is the case because the method would have already returned otherwise. This is sufficient for the prover to be able to show that the method returns `-1` if the loop continues till the loop guard becomes false rather than exiting early, and the code itself is enough to verify the other case of exiting early.

Listing 8: Specification of the `findIndex` method

```
1       /*@ public normal_behaviour
2         @ ensures contains(key) ==> 0 <= \result < size && keys[\result] == key;
3         @ ensures !contains(key) ==> \result == -1;
4         @*/
5       private int /*@ pure @*/ findIndex(int key) {
6           /*@ loop_invariant 0 <= i && i <= size
7             @  && (\forall int x; 0 <= x && x < i; keys[x] != key);
8             @ assignable \nothing;
9             @ decreases size - i;
10            @*/
11          for(int i = 0; i < size; i++) {
12              if(keys[i] == key) {
13                  return i;
14              }
15          }
16          return -1;
17      }
```

The specifications of get, remove and other methods are omitted as they are comparable to what we have discussed for `put` in this section. They can still be found in the appendix in Listing 10.

4.2.3 Process

We started with trying to verify a hash map that used separate chaining for handling collisions. We tried guiding the prover where it got stuck and read the textbook to understand new things encountered but could not make much progress. After a few weeks, we took the advice of our peers and switched to verifying a simple key store consisting of a single array of type `Entry` storing key-value pairs and performing the three main operations by doing linear scans. None of its methods could be verified automatically. The incomplete proof trees led us to believe that the problem might be in how we were handling Objects. We referred to a study on verifying OpenJDK's IdentityHashMap [4] and immediately found that they used `==` for testing

equality rather than `equals`. By changing this, the pure methods `get` and `contains` were now automatically verifiable, but `put` and `remove` were not.

We noticed later that the paper also compared the performance of KeY in verifying implementations of hash maps that differed in how they handled collisions (separate chaining or linear probing) and the types of keys and values supported (int-int, int-Object, Key-Object using `==` for equality and Key-Object using `equals` for equality, where Key is a class with an immutable field for storing an Integer key), and reported that the prover was unable to automatically verify the correctness of put and remove in the implementations using non-primitive keys. We also noted that they used separate arrays for keys and values rather than a class like `Entry`. Adapting our implementation did not result in improvements and the prover still got stuck at similar stages of the proof.

Another notable difference between our implementation and theirs was that they relied on a `findIndex` method, which makes sense for a hash map, to perform all three operations, and also used it in their specifications, whereas we used the `contains` method and loops for the implementations, and more importantly used `\exists` to talk about the index at which the entry to be operated on is located in the specifications. Replacing the quantifier with `findIndex` in the specifications resulted in all operations being proven by KeY automatically. By also changing `contains` to a model method, we arrive at the implementation we just described in the previous sub-sections.

5 Discussion

We had set out to write this paper with the goals of providing an intuitive understanding of the working of KeY, and explaining what it means to prove something to be correct using it. Section 2 and Section 4 have been written with the first goal in mind. Section 5.1 discusses the meaning of correctness and explains the kind of guarantees KeY provides and the assumptions it relies on to do so. Section 5.2 talks about how the principle of Design by Contract makes programs easier to verify, Section 5.3 is on the complementary nature of testing and formal verification and Section 5.4 discusses the intended approach for using KeY and how we used it in the case studies. Finally, Section 5.5 summarizes the capabilities and limitations of using KeY in practice.

5.1 Meaning of Correctness, Guarantees and Assumptions

It should be quite evident at this point that the prover only verifies that the implementation adheres to the provided specification, meaning that the criteria for correctness have to be defined by the programmer. There are, however, several assumptions made by KeY that impact the meaning of correctness. For example, KeY assumes that Objects are not allowed to be `null` by default and need an explicit `nullable` annotation. Both the case studies rely on this, as the sorting algorithm does not perform null checks on the array passed as input, and the get method of the key store returns `null` to indicate the absence of a key without having to address the case where a key is actually mapped to `null`. Other assumptions are for instance the integer semantics used (overflow checking), allowing exceptions to be thrown and caught, and assumptions about the memory model.

Since correctness depends on the specifications, the kind of guarantees KeY can provide are whatever can be expressed in specifications. JML can be used to express arbitrarily complex functional properties for sequential Java 7 code, but it has also been used to express non-functional properties like information flow and resource usage with the help of constructs for

defining memory locations accessible in code, and ghost variables to for instance count the number of calls made to a method.

5.2 Verifiable Code

Doing the case studies has made it clear to us that just like there is a notion of *testable* code, there is also *verifiable* code. We gave an example in Section 4.2.3 where we talked about how using the method `findIndex` in our specifications instead of using `\exists` quantifiers simplified our specifications significantly and allowed the prover to start proving all methods of the key store automatically. This clearly highlighted how some specifications are a lot more amenable to verification than others, despite being equivalent in meaning. Another example of this was seen in how we split the implementation of Insertion Sort into three methods, allowing us to prove a part of the algorithm and focus our efforts on the remaining unproven parts. In general, deductive verifiers benefit from modularized programs following the principles of Design by Contract. The reason for this is that verifying smaller methods is a lot more computationally feasible, and because the contracts are used instead of the bodies of called methods, which can provide valuable information to guide provers in their search for proofs for larger pieces of software relying on other methods.

5.3 Automated Formal Verification vs Testing

Our experience verifying a very simple hash method revealed that automated formal verification cannot play the role of testing. The hash method `int hash(int key)` was implemented to simply return `key % a.length` where `a` was the bucket array, and we wished to ensure that the value returned by the method always lies between `0` and `a.length`. However, we were simply unable to prove this. After many hours, we thought to actually test the method and found that `m % n` returns a negative number in Java if `m` is negative and `n` is positive.

To prove the correctness of a program, it (and its specification) must be correct in the first place. Quicker approaches with better feedback like testing should be used to gain confidence in a program before attempting to prove its correctness. Other studies using KeY have also reported on using this approach [4]. We also note that several of the papers attempting to verify software ended up finding bugs in them [5], [6], [8].

5.4 The Design of KeY and how we used it

KeY was designed with the verification of complex real-world Java programs in mind, in a time when computation power was much more limited. Even today, automated provers are limited by computation power required due to an explosion in the proof search space with increasing complexity of programs. As a result, KeY is more like an interactive proof assistant but with support for automating a lot of the tedious work in proofs. Combining the intuition of the user and the automation of the prover have allowed it verify or find bugs in complex pieces of software.

The intended approach for using KeY is to run the automated prover, and if it gets stuck, inspect the attempted proof and either guide the prover to finish the proof or adjust the implementation and specification and try again. We attempted to follow this approach for the first few weeks of the study but made limited progress. This was due to a combination of lack of experience to be able to quickly identify relevant information from the attempted proof and a lack of knowledge on what the right rules to guide the prover might be or what the implications of some of the information presented in the interface are.

In the later half of the study, we reduced the time spent trying to finish proofs interactively and instead took on a trial and error approach. We tried to quickly identify which clauses might be problematic by looking at the branching of the proof tree and the highlighted parts of the specification also shown in the interface, and then adjust the implementation or specification. This was done taking inspiration from a peer doing a similar study with Dafny, which gave feedback by simply highlighting which conditions could not be proven. This approach allowed us to arrive at automatically verifiable code more often, but is a sub-optimal way of using KeY.

5.5 Capabilities and Limitations of KeY

The biggest strength of KeY is also its biggest drawback. It is very capable of verifying complex software but is itself quite complex with a steep learning curve in order to achieve this. Another study by non-expert users of KeY attempting to verify parts of the OpenJDK's API identifies problems like a requirement of expertise in the underlying proof theory, challenges in identifying whether a proof cannot be closed because of a mistake in the specification or in the implementation and the large impact of the choice of the prover configuration on the performance of the prover [7], matching our experience. We also observed that most of the studies verifying complex software with KeY include the developers of KeY as co-authors, and the usage of the tool appears to be limited to a handful of people. As a result, availability of documentation and support is also limited. A question we asked on StackOverflow about proving the swap method for sorting is still unanswered after several weeks[1].

For writing and verifying new software, other tools might be more easy to use. For instance, Ada/SPARK is used in industry for writing verified software and is quite mature. A list of deductive verification tools can be seen in Table 1. However, there are no comparable tools to KeY for deductive verification of Java software, especially software that cannot be refactored to be made easier to verify. Other tools do exist for other kinds of formal verification of Java and are mentioned in Section 7 on related work.

6 Responsible Research

There are two main ethical aspects to this study – reproducibility and societal impact, covered in Section 6.1 and Section 6.2.

6.1 Reproducibility

All code presented in this paper is available in a public GitHub repository[2] under an MIT license. The repository provides a full provenance of how the code came to its present state, and the code from attempts at verifying different implementations in the case studies that failed have also been left in the repository. It also provides instructions for replicating the proofs. Key decisions made in performing the case studies are documented in the sub-sections on process in Section 4. The version of KeY used is 2.12.3. No plugins for KeY were used. No datasets were used or produced in the course of this study.

[1] StackOverflow post: https://stackoverflow.com/questions/79656775/key-prover-failing-to-verify-that-swapping-two-elements-of-an-array-results-in-a

[2] link: https://github.com/Tejas-Kochar/key-verifier/

6.2 Societal Impact

Formal verification has significant impact on society due to its use in critical systems for providing correctness guarantees. KeY itself has been used to verify the correctness of real-world software like that used for tallying results in Norwegian elections or parts of the OpenJDK library which are depended upon by applications running on billions of devices. However, these were done by researchers to demonstrate the capabilities of the tool. The usage of KeY otherwise appears to be quite limited due to a variety of reasons which significantly reduces its societal impact in practice compared to some other formal verification tools that are used for critical systems in industry. This paper aims to make it easier to understand the tool and highlight its limitations and capabilities, thus making the tool more accessible and reducing the risk of incorrect usage.

7 Related Work

This study is one of five such studies, each aiming to understand an automated program verification tool. The over-arching goal is to get a sense of the kind of correctness guarantees the tools give and the kind of technologies they use to prove correctness. Other work has been done in similar directions. Hähnle and Huisman give an overview of the field of deductive verification, including a very nice summary of the underlying theory and an account on the evolution of tools [16]. Boerman et. al perform a case study on the syntactic and semantic differences in how KeY and OpenJML treat JML specifications with a goal of gaining insights for interoperability of tools on the same specifications [22]. Software Verification competitions are held annually to compare different verification tools and exchange new techniques and advancements [23].

A non-exhaustive list of actively maintained deductive verification tools is given in Table 1 in Appendix B. All the tools in the list are capable of static verification. Some tools like VerCors, VeriFast and SPARK/Ada can check for memory safety. We also note that Boogie, Viper and Why3 are used as backends for a large number of other tools. Hähnle and Huisman discuss a subset of the tools in the list in greater detail [16] but miss more recent tools based on Viper and tools like Stainless and Cameleer that verify more niche languages (Scala, OCaml).

Some other notable tools relevant for Java include the likes of Infer and TLA+. Infer performs fully automated deductive verification based on separation logic to verify the memory safety of C programs [24]. It has been extended to automatically check for a wide range of bugs in languages like Java, C, C++ by static analysis and is a part of Meta's CI pipeline[3]. TLA+, on the other hand, is a language for formally specifying distributed algorithms which can then be checked by model checkers [25]. Amazon has used it to specify and check their designs for industrial distributed systems like DynamoDB implemented in Java [26].

Several case studies have been done with KeY. Kroening and Păsăreanu describe its usage to find a bug and verify a fixed implementation of an implementation of TimSort, used as the standard sorting algorithm in OpenJDK [5]. Knüppel et. al share their perspective of verifying parts of OpenJDK's API as non-expert users of KeY [7]. Hiep et. al verified OpenJDK's LinkedList data structure and the bugs found in the attempt [6]. Finkbeiner and Kovács attempted to verify and found bugs in an implementation of the in-place super scalar sample sort algorithm [3], and similarly Herber and Wijs found bugs in OpenJDKs BitSet class [8].

[3]Website:https://fbinfer.com/docs/about-Infer

8 Conclusions and Future Work

KeY is a powerful tool but has a very small user base and a steep learning curve. Verifying the correctness of a program with it means proving that the program adheres to its specifications. Its specifications can be used to express functional and even some non-functional properties using the expressive Java Modeling Language (JML). However, one is required to have an know the assumptions of JML and KeY which can often blindside new users. It is hard to get feedback on where problems are from failed attempts of proofs as they require inspecting the proof tree and the current state of the proof. This is a significant disadvantage compared to approaches like testing that provide easy-to-understand feedback fast, as a result of which we only see formal verification complementing testing.

The goal of this paper is to give an intuition of how the tool works and what it is like to use it in terms of its limitations and capabilities. This naturally abstracts away a lot of details. Having had an intuition of how the tools work, it is perhaps now even more important to know which aspects have not been covered. This paper does not touch upon the concepts of Symbolic Execution, Framing, Algebraic Data Types, Taclets, helper functions, the memory model, the use of access specifiers, handling of generics and more. A comprehensive treatment of these can be found in the textbook [1]. Even more importantly, we do not touch upon the several plugins and tools in the KeY ecosystem, for instance the Symbolic Execution Debugger, test case generation and Eclipse plugins. KeY is due for a major version upgrade and its latest beta versions support user defined ADTs, functions, a new in-built ADT for modelling maps, etc. We expect these features to make it a lot easier for the automated prover to verify things, making case studies like this much easier.

This study was quite limited in its duration and scope and a lot could be done to expand on them. Future work could investigate the many tools being built around KeY or explore the support for newer features like concurrency and floating point operations. We also see a lot of potential in using formal verification for generating code using LLMs by first defining a formal specification and then checking the correctness of generated code against it.

A Code Listings

All code made in this study is available in a public GitHub repository: https://github.com/Tejas-Kochar/key-verifier/

Listing 9: Complete specifications for `shift` and `swap` helper methods for Insertion Sort

```
    /*@ normal_behaviour
      @ requires 1 <= i < a.length;
      @ requires (\forall int x; 0 <= x < i-1; a[x] <= a[x+1]);
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ ensures (\forall int x; 0 <= x < i; a[x] <= a[x+1]);
      @ assignable a[0..i+1];
      @*/
    void shift(int[] a, int i) {
        /*@
          @ loop_invariant -1 <= j < i;
          @ loop_invariant \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
          @ loop_invariant (\forall int x; j+1 <= x < i; a[x] <= a[x+1]);
          @ loop_invariant (\forall int x; 0 <= x < j; a[x] <= a[x+1]);
          @ decreases j+1;
          @ assignable a[0..i+1];
          @*/
        for (int j = i-1; j >= 0 && a[j] > a[j+1]; j --) {
            swap(a, j, j+1);
        }
    }

    /*@ normal_behaviour
      @ requires 0 <= j < a.length && 0 <= j1 < a.length;
      @ ensures \old(a[j]) == a[j1] && \old(a[j1]) == a[j];
      @ ensures \dl_seqPerm(\dl_array2seq(a), \old(\dl_array2seq(a)));
      @ assignable a[j], a[j1];
      @*/
    void swap(int[] a, int j, int j1) {
        int tmp = a[j];
        a[j] = a[j1];
        a[j1] = tmp;
    }
```

Listing 10: Complete implementation and specification of `get` and `remove` methods of `KeyStore`

```
    /*@
      @ public normal_behaviour
      @ ensures contains(key) ==> \result == values[findIndex(key)];
      @ ensures !contains(key) ==> \result == null;
      @ assignable \nothing;
    */
    public Object /*@ pure @*/ get(int key) {
        int index = findIndex(key);
        if(index == -1) {
            return null;
        }
        else {
            return values[index];
        }
    }

    /*@
      @ public normal_behaviour
      @ requires contains(key);
```

```
20        @ ensures !contains(key);
21        @ ensures size == \old(size) - 1;
22        @ ensures (\forall int x; 0 <= x < size;
23        @         (keys[x] == \old(keys[x]) && values[x] == \old(values[x]))
24        @         || (keys[x] == \old(keys[size-1]) && values[x] ==
              \old(values[size-1])));
25        @
26        @ also
27        @
28        @ public normal_behaviour
29        @ requires !contains(key);
30        @ ensures !contains(key);
31        @ assignable \nothing;
32        @*/
33      public void remove(int key) {
34          int index = findIndex(key);
35          if (index != -1) {
36              size --;
37              keys[index] = keys[size];
38              values[index] = values[size];
39          }
40      }
```

B Tables

Table 1: Actively maintained deductive verification tools (Tool), the programming languages they can verify (Target PL) and tools used internally for verification (Backend)

Tool	**Target PL**	**Backend**
AutoProof	Eiffel	Boogie
Boogie	BoogiePL	Z3, other SMT solvers
Cameleer	OCaml	Why3
Dafny	Dafny	Boogie
Frama-C	C	Why3
Gobra	Go	Viper
KeY	Java	KeY prover, SMT solvers
Nagini	Python	Viper
OpenJML	Java	SMT solvers
Prusti	Rust	Viper
SPARK/Ada	Ada	GNATprove, Why3
Stainless	Scala	Z3, CVC4, CVC5
VCC	C	Boogie
VerCors	Java, C, PVL	Viper
VeriFast	C, Java	Z3, other SMT solvers
Viper	Silver	Boogie
Why3	WhyML, MicroPython, micro C, OCaml	SMT Solvers, ATPs

References

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt and M. Ulbrich, *Deductive Software Verification – The KeY Book*, 1st ed., W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt and M. Ulbrich, Eds. Springer International Publishing, 2016, pp. XXXII–702, ISBN: 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6.

[2] H. T. Klev, "Verifying eva: Formal verification of the software for deciding norwegian governmental elections," M.S. thesis, University of Oslo, 2020.

[3] B. Beckert, P. Sanders, M. Ulbrich, J. Wiesler and S. Witt, "Formally Verifying an Efficient Sorter," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., vol. 14570, Cham: Springer Nature Switzerland, 2024, pp. 268–287, ISBN: 9783031572456 9783031572463. DOI: 10.1007/978-3-031-57246-3_15.

[4] M. De Boer, S. De Gouw, J. Klamroth, C. Jung, M. Ulbrich and A. Weigl, "Formal Specification and Verification of JDK's Identity Hash Map Implementation," *Formal Aspects of Computing*, vol. 35, no. 3, pp. 1–26, Sep. 2023, ISSN: 0934-5043, 1433-299X. DOI: 10.1145/3594729.

[5] S. De Gouw, J. Rot, F. S. De Boer, R. Bubel and R. Hähnle, "Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., vol. 9206, Cham: Springer International Publishing, 2015, pp. 273–289, ISBN: 9783319216898 9783319216904. DOI: 10.1007/978-3-319-21690-4_16.

[6] H.-D. A. Hiep, O. Maathuis, J. Bian, F. S. De Boer and S. De Gouw, "Verifying OpenJDK's LinkedList using KeY (extended paper)," *International Journal on Software Tools for Technology Transfer*, vol. 24, no. 5, pp. 783–802, Oct. 2022, ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-022-00679-7.

[7] A. Knüppel, T. Thüm, C. Pardylla and I. Schaefer, "Experience Report on Formally Verifying Parts of OpenJDK's API with KeY," *Electronic Proceedings in Theoretical Computer Science*, vol. 284, pp. 53–70, Nov. 2018, ISSN: 2075-2180. DOI: 10.4204/EPTCS.284.5.

[8] A. S. Tatman, H.-D. A. Hiep and S. De Gouw, "Analysis and Formal Specification of OpenJDK's BitSet," in *Integrated Formal Methods*, P. Herber and A. Wijs, Eds., vol. 14300, Cham: Springer Nature Switzerland, 2024, pp. 134–152, ISBN: 9783031477041 9783031477058. DOI: 10.1007/978-3-031-47705-8_8.

[9] W. Ahrendt, W. Mostowski and G. Paganelli, "Real-time Java API specifications for high coverage test generation," en, in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, Copenhagen Denmark: ACM, Oct. 2012, pp. 145–154, ISBN: 9781450316880. DOI: 10.1145/2388936.2388960.

[10] J. Bian, H.-D. A. Hiep, F. S. De Boer and S. De Gouw, "Integrating ADTs in KeY and their application to history-based reasoning about collection," *Formal Methods in System Design*, vol. 61, no. 1, pp. 63–89, Aug. 2022, ISSN: 0925-9856, 1572-8102. DOI: 10.1007/s10703-023-00426-x.

[11] M. Huisman and W. Mostowski, "A Symbolic Approach to Permission Accounting for Concurrent Reasoning," in *2015 14th International Symposium on Parallel and Distributed Computing*, Limassol, Cyprus: IEEE, Jun. 2015, pp. 165–174, ISBN: 9781467371476 9781467371483. DOI: 10.1109/ISPDC.2015.26.

[12] W. Mostowski and M. Ulbrich, "Dynamic Dispatch for Method Contracts Through Abstract Predicates," in *Transactions on Modularity and Composition I*, S. Chiba, M. Südholt, P. Eugster, L. Ziarek and G. T. Leavens, Eds., vol. 9800, Cham: Springer International Publishing, 2016, pp. 238–267, ISBN: 9783319469683 9783319469690. DOI: 10.1007/978-3-319-46969-0_7.

[13] H.-D. A. Hiep, J. Bian, F. S. De Boer and S. De Gouw, "A Tutorial on Verifying LinkedList Using KeY," in *Deductive Software Verification: Future Perspectives*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle and M. Ulbrich, Eds., vol. 12345, Cham: Springer International Publishing, 2020, pp. 221–245, ISBN: 9783030643539 9783030643546. DOI: 10.1007/978-3-030-64354-6_9.

[14] B. Beckert, R. Hähnle, M. Hentschel and P. H. Schmitt, "Formal Verification with KeY: A Tutorial," in *Deductive Software Verification – The KeY Book*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt and M. Ulbrich, Eds., vol. 10001, Cham: Springer International Publishing, 2016, pp. 541–570, ISBN: 9783319498119 9783319498126. DOI: 10.1007/978-3-319-49812-6_16.

[15] W. Ahrendt, B. Beckert, R. Hähnle, P. Rümmer and P. H. Schmitt, "Verifying Object-Oriented Programs with KeY: A Tutorial," in *Formal Methods for Components and Objects*, D. Hutchison, T. Kanade, J. Kittler *et al.*, Eds., vol. 4709, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 70–101, ISBN: 9783540747918 9783540747925. DOI: 10.1007/978-3-540-74792-5_4.

[16] R. Hähnle and M. Huisman, "Deductive software verification: From pen-and-paper proofs to industrial tools," in *Computing and Software Science*, B. Steffen and G. Woeginger, Eds., vol. 10000, Cham: Springer International Publishing, 2019, pp. 345–373, ISBN: 9783319919072 9783319919089. DOI: 10.1007/978-3-319-91908-9_18.

[17] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259.

[18] R. W. Floyd, "Assigning Meanings to Programs," in *Proceedings of Symposium on Applied Mathematics*, vol. 19, 1967, pp. 19–32.

[19] V. R. Pratt, "Semantical considerations on floyd-hoare logic," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 1976, pp. 109–121. DOI: 10.1109/SFCS.1976.27.

[20] G. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, Jul. 2004. DOI: 10.1016/j.jlap.2004.05.001.

[21] E. W. Dijkstra, *A discipline of programming*, English, Prentice-Hall Series in Automatic Computation. Englewood Cliffs, N.J.: Prentice-Hall, Inc. XVII, 217 p. 1976.

[22] J. Boerman, M. Huisman and S. Joosten, "Reasoning about JML: Differences between KeY and OpenJML," in *Integrated Formal Methods*, C. A. Furia and K. Winter, Eds., vol. 11023, Cham: Springer International Publishing, 2018, pp. 30–46, ISBN: 9783319989372 9783319989389. DOI: 10.1007/978-3-319-98938-9_3.

[23] D. Beyer and J. Strejček, "Improvements in software verification and witness validation: SV-COMP 2025," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Gurfinkel and M. Heule, Eds., vol. 15698, Cham: Springer Nature Switzerland, 2025, pp. 151–186, ISBN: 9783031906596 9783031906602. DOI: 10.1007/978-3-031-90660-2_9.

[24] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann and R. Joshi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465, ISBN: 978-3-642-20398-5.

[25] L. Lamport, J. Matthews, M. Tuttle and Y. Yu, "Specifying and verifying systems with TLA+," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ser. EW 10, New York, NY, USA: Association for Computing Machinery, 1st Jul. 2002, pp. 45–48, ISBN: 978-1-4503-7806-2. DOI: 10.1145/1133373.1133382.

[26] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker and M. Deardeuff, "How amazon web services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 23rd Mar. 2015, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2699417.