Formal Specification and Verification of JDK's Identity Hash Map Implementation

Martin de Boer[1], Stijn de Gouw[1], Jonas Klamroth[2], Christian Jung[3], Mattias Ulbrich[3][0000−0002−2350−1831], and Alexander Weigl[3][0000−0001−8446−4598]

[1] Open University, Heerlen, The Netherlands
[2] FZI Research Center for Information Technology, Karlsruhe, Germany
[3] Karlsruhe Institute of Technology, Karlsruhe, Germany

Abstract. Hash maps are a common and important data structure in efficient algorithm implementations. Despite their wide-spread use, real-world implementations are not regularly verified.
In this paper, we present the first case study of the `IdentityHashMap` class in the Java JDK. We specified its behavior using the Java Modeling Language (JML) and proved correctness for the main insertion and lookup methods with KeY, a semi-interactive theorem prover for JML-annotated Java programs. Furthermore, we report how unit testing and bounded model checking can be leveraged to find a suitable specification more quickly. We also investigated where the bottlenecks in the verification of hash maps lie for KeY by comparing required automatic proof effort for different hash map implementations and draw conclusions for the choice of hash map implementations regarding their verifiability.

Keywords: Real-world case study · Deductive Program Verification · Java Modeling Language · Verified Hash Map · Verified Data Structure · Cooperative Verification

1 Introduction

Maps are versatile data structures and a common foundation for important algorithms as they provide a simple modifiable association between two objects: the key and a value. A hash map realizes this association with a (constant time) hash function, which maps a key to a memory location in the managed memory space. Thus, the typical operations, i.e., lookup, update and deletion of associations, achieve a constant run-time on average.

To optimize their performance, hash maps require complex memory layout and collision resolution strategies. The memory layout describes where and how associations are stored. The collision strategy handles the location resolution when the memory location is already occupied by a different key with the same hash. Further, an implementation needs to decide when and how a restructuring of the memory layout is necessary to maintain the performance over time because the addition and removal of association leads to fragmentation.

In this paper, we present the specification and verification of the `Identity-HashMap` class of the Java SDK as it appears in the latest update of JDK7 and newer JDK versions (up to JDK17)[4]. To our knowledge, this is the first case study, which formally verifies a real-world hash map implementation from a mainstream programming language library. In particular, it is part of the Java Collections Framework, which is one of the most widely used libraries. We formally specify the behavior of the implementation using the Java Modeling Language JML. The case study with all artifacts is available at [5]. We show how we combined various JML-based tools (OpenJML, JJBMC, and KeY) together to exploit their strengths and avoid the weaknesses. In detail, we firstly used JUnit tests with generated runtime assertion and JJBMC [2] to quickly prove contracts and strengthened the specification, OpenJML [7] to automatically prove contracts, and finally KeY [1] to provide preciseness by the cost of performance and required user interaction. Finally, we were able to prove 15 methods of the class with KeY.

Furthermore, we describe how various implementation choices of hash maps affect the verification performance with KeY. For this, we re-implemented commonly used hash map concepts in Java and specified them with JML.

Related Work. The hash map/table data structure of a linked list has been studied mainly in terms of pseudo code of an idealized mathematical abstraction, see [15] for an Eiffel version and [16] for an OCaml version. Hiep et al. [10] and Knüppel et al. [11] investigate correctness of some other classes of the Collections framework using KeY, the latter mainly as a “stepping stone towards a case study for future research”. In [4], the authors specify and verify the Dual Pivot Quicksort algorithm (part of the default sorting implementation for primitive types) in Java.

2 Preliminaries

The Java Modeling Language (JML) [13] is a behavioral interface specification language [9] for Java programs according to the of design-by-contract paradigm [14]. Lst. 1 shows an excerpt of the specification for the hash map method `get`; the full specification is covered in detail in Sect. 4. JML annotations are enclosed in comments beginning with `/*@` or `//@`. The listing contains a *method contract* (lines 5-10) covering the *normal behavior case* in which an exception must not be thrown. The `requires` and `ensures` clauses specify the pre- and postcondition respectively; the framing condition is given in the `assignable` clause which lists the heap locations modifiable by the method. The special keyword `\nothing` indicates that no existing heap location must be modified, but new objects may be allocated. `\strictly_nothing` specifies that the heap must not be modified at all. Multiple contracts for a method are separated with the keyword `also`. JML also supports *class invariants* (line 3) which need to be established before and after every method invocation. To conduct inductive proofs for loops, these can be annotated with *loop specifications* (lines 19-22). The *loop invariants* (`maintains`) must hold when the loop is reached and after

[4] http://hg.openjdk.java.net/jdk7u/jdk7u/jdk/file/4dd5e486620d/src/share/classes/java/util/IdentityHashMap.java

```
 1 class IdentityHashMap {
 2   private /*@ nullable */ Object[] table;
 3   //@ public invariant table != null;
 4 
 5   /*@ public normal_behavior
 6     @   requires (\exists \bigint i; 0 <= i < table.length/(\bigint)2;
 7     @       table[i*2] == maskNull(key));
 8     @   ensures  (\exists \bigint i; 0 <= i < table.length/(\bigint)2;
 9     @       table[i*2] == maskNull(key) && \result == table[i*2+1]);
10     @   assignable \nothing;
11     @ also public normal_behavior ...
12     @*/
13   public /*@ nullable */ Object get(/*@ nullable */ Object key) {
14     Object k = maskNull(key); Object[] tab = table;
15     int len = tab.length, i = hash(k, len);
16 
17     //@ ghost \bigint hash = i;
18     /*@ // Index i is always an even value within the array bounds
19       @ maintaining 0 <= i < len && i % (\bigint)2 == 0;
20       @ maintaining ...
21       @ decreasing hash > i ? hash - i : hash + len - i;
22       @ assignable \strictly_nothing;
23       @*/
24     while (true) {
25       Object item = tab[i];
26       if (item == k)     return tab[i+1];
27       if (item == null)  return null;
28       i = nextKeyIndex(i, len);
29     }
30   }
31 }
```

Listing 1. The lookup method `get` of class `IdentityHashMap` as an introductory example of JML specifications.

every iteration. In the example, the variable `i` is specified to remain in range between 0 and `len` and is always even. The *loop variant* expression (`decreasing`) computes to a natural number which must be strictly decreased in every loop iteration. The `assignable` clause specifies the heap locations all loop iterations are allowed to change. JML extends the Java expression language by first-order logic constructs like existential (`\exists`) and universal quantification (`\forall`). Also, the construct `(\num_of int x;` G`;` C`)` is relevant for the case study. It counts the number of values for `x` such that the guard G and the condition C are satisfied. For instance, `(\num_of int i; 0<=i<a.length; a[i] != null)` returns the number of non-null elements in array `a`. The identifier `\result` refers to the method's return value in postconditions, and the expression `\old(`E`)` evaluates the expression E in the pre-state of the method invocation. JML `ghost` variables (line 17) behave like local Java variables during verification, but are not available at runtime and must therefore not have an impact on the effects and result of the method they are declared in. The special primitive JML value type `\bigint` refers to the mathematical integers $\mathbb{Z}$.[5] Finally, JML adds a few

[5] At various places in the specifications, explicit casts like `(\bigint)2` have been added. These force the semantics of surrounding arithmetic operations to be in $\mathbb{Z}$ (rather than in 32-bit `int` with overflows) which simplifies the verification considerably.

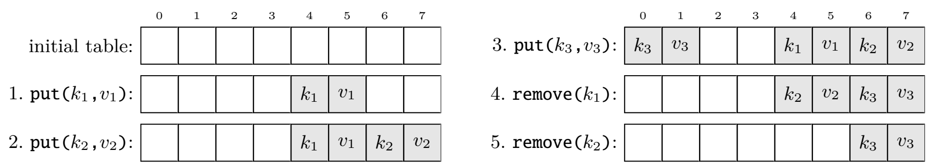

0 1 2 3 4 5 6 7
initial table:
1. put(k_1,v_1): k_1 v_1
2. put(k_2,v_2): k_1 v_1 k_2 v_2
3. put(k_3,v_3): k_3 v_3 k_1 v_1 k_2 v_2
4. remove(k_1): k_2 v_2 k_3 v_3
5. remove(k_2): k_3 v_3

Fig. 1. Memory layout of the `table` array with length $N = 8$, $h_i = \text{hash}(k_i, N)$ for hashes $h_1 = 4$, $h_2 = 4$, and $h_3 = 6$.

modifiers to the language like **`nullable`** which specifies that a field, method argument or return value may be **`null`**. Without the modifier, the value must not be **`null`** and, in the case of arrays, must not contain **`null`** values.

JML specifications can be used in different formal analyses, ranging from formal documentation, test case generation, runtime assertion checking to deductive verification. This paper will focus on the deductive verification of JML-annotated programs using two tools implementing different deductive JML verification approaches: KeY and JJBMC.

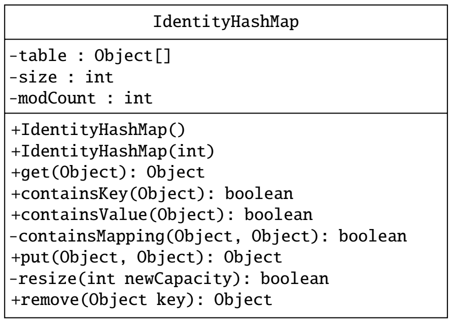
KeY is a theorem prover for JML-annotated Java programs that supports automatic and interactive verification. KeY encodes method contracts as proof obligations in dynamic logic, a program logic similar to the weakest precondition calculus or Hoare logic. The programs inside the dynamic logic formulas are resolved by applying a series of inference rules, thus symbolically executing the code and hence producing the weakest preconditions in first order logic. Further inference rules are applied to discharge these resulting obligations. KeY possesses a powerful automatic strategy that can prove most obligations fully automatically. In case of more sophisticated heavyweight specifications (like the ones in the present hash map case study), the user can apply inference rules interactively to guide the proof.

The tool JJBMC [2] on the other hand combines modular deductive verification with bounded model checking. It translates JML specifications to Java statements using additional assumptions and assertions. The enriched Java sources are then submitted to the state-of-the-art Java bounded model checker JBMC [8]. In Sect. 5.1 we will report how the combination of bounded verification with modular concepts inside JJBMC helped engineering the specifications.

3 The Verification Subject: JDK's IdentityHashMap

The `IdentityHashMap` is a hash table implementation of the interface `java.util.Map` of the Java Collections Framework. Fig. 2 shows an overview of the class. Like any `Map`, it implements a modifiable mapping between keys and values, s. t. every key k_i is associated with exactly one value v_i. In the `IdentityHashMap`, two keys k_1 and k_2 are considered equal if and only if $k_1 = k_2$ (equality by reference, see Lst. 1 line 26 and Lst. 4 line 33). In contrast, the equality of keys in `HashMap` is defined by the `equals` method).

The `IdentityHashMap` stores the key-value entries sequentially in a one-dimensional array (private field `table`). The class relies on the built-in function `System.identityHashCode(o)` which returns a hash code for the object `o`. The hash is the first candidate spot in `table` to lookup the entry, or locating a free spot to store the entry.



```
IdentityHashMap
-table : Object[]
-size : int
-modCount : int
+IdentityHashMap()
+IdentityHashMap(int)
+get(Object): Object
+containsKey(Object): boolean
+containsValue(Object): boolean
-containsMapping(Object, Object): boolean
+put(Object, Object): Object
-resize(int newCapacity): boolean
+remove(Object key): Object
```

Fig. 2. Excerpt of the `IdentityHashMap` class.

When an entry (k_1, v_1) is added (`put`(k_1, v_1), cf. Lst. 4), a hash $h_1 \in \{0, 2, 4, \ldots, N-2\}$ is calculated based on the hash of the key k_1 and the length N of the `table` (line 28). The key k_1 is then stored in `table` at the (even) index h_1, and the value v_1 is stored adjacently at (odd) index $h_1 + 1$ (line 40). Item 1. in Fig. 1 shows the case where an entry is added to an empty map. In case k_1 was already present in the table, it would not be inserted a second time (this would break uniqueness), but its associated value would be overwritten with v_1. While keys are unique, there is no guarantee that their *hash values* are. Collisions might occur: the calculated index in `table` can be already occupied by an entry with a different key. The new entry is then stored at the next free position in `table` (item 2. in Fig. 1, where (k_2, v_2) is stored at index 6, while its hash h_2 is 4). If that index idx is taken as well, the next even index $idx' = idx + 2 \mod N$ is calculated by `nextKeyIndex` and tried, until a free spot is found (item 3. in Fig. 1). This ensures that there is no gap (empty slots) between the calculated index and the actual index of a key. This collision resolution strategy is called *linear probing* [12]. Sect. 6.1 discusses other strategies. `IdentityHashMap` supports using the `null` value as a key. To distinguish the `null` key from an empty slot in the table, a constant object reference, `NULL_KEY`, is used in place of `null`.

The `get(k)` (Lst. 1) method retrieves the value for a given key k. It searches the table with the same process that we described above for insertion: start at the hash of k (line 15+25) and move to the next key slot (two spots further, modulo N, see line 28) until k is found (line 26). The search also terminates when an empty element in the array is encountered: this means there is no entry with key k (line 27). To ensure termination, it is thus crucial that the array at all times contains at least one empty slot.

We do not discuss removing an entry (method `remove`) in detail, but only note that `table` needs to be rearranged as if the entry had never been added in the first place, so that `remove` introduces no gaps between the calculated and actual index of a key. For an example, see last items in Fig. 1.

4 Specification and Verification of the IdentityHashMap

We now discuss the specification and verification of core parts of the `IdentityHashMap`. The full case study comprises several hundred lines of source code and specifications and over 1.4 million proof steps (Tab. 1). An exhaustive exposition is therefore clearly not feasible. Instead, we focus on the core methods and highlight several of the main proof obligations and their proofs in this section.

Particularly with case studies of such a large size, it can be challenging, but is crucial, to make and keep the formal specifications manageable and understandable. Developers of the specification must quickly see which properties were formalized already and which remain to be fixed or added (if they turn out to be flawed during analysis). During the proof, one must understand the specifications sufficiently well to use them in proving the verification conditions. Clients of the `IdentityHashMap` should be able to use the class solely on the basis of the specifications (without looking at particular implementation details). To facilitate understandability, our specifications include comments in natural language that explain what the formal property expresses.

Some of the core properties maintained by the class invariant are for example that the table contains at least one empty spot (so that lookup methods terminate) (line 18 in Lst. 2) and that all spots between the hash value and the actual index (including the wrap-around behavior as described in Sect. 3) in the table are occupied (lines 24 and 34 in Lst. 2).

One could use the pure `hash` method from the Java code in the class invariant to refer to the hash of an object. But this can be inconvenient for the proof process: the `hash` method body must then be executed to derive that heap modifications do not alter hashes of existing objects (and that the result is deterministic, etc). We simplify this by introducing a new mathematical (deterministic) function `dl_genHash` that does not rely on the heap to refer to an object hash and adding a postcondition to `hash` that its return value is `dl_genHash`. Let us now discuss some of the main proof obligations that arise in the verification of this class.

Termination of get(..). Lst. 3 shows the specification of the loop in the `get` method. This loop also appears (in slightly different forms) in many other core methods of the hash map: the three `contains*` methods, `put`, and `remove`. The main goal of this loop is to search for a given key. As the loop guard is *true*, the loop only terminates if a `return` statement is encountered. Intuitively, if the given key is not present, the loop eventually hits the empty spot in the `table`, which the class invariant ensures to exist. If the key *is* present, eventually the condition `item == k` becomes *true*.

We now prove termination formally, using the variant in the `decreasing` clause (line 21 in Lst. 1). Suppose the loop invariant and the loop guard hold at the start of a loop iteration. If a `return` statement is hit in the iteration, then clearly the loop terminates promptly. Otherwise, we must show that the variant has decreased at the end of the iteration (with an updated value of `i`), but remains non-negative. The following cases (where `i` is the value at the start

```
/*@ public invariant table != null &&
  @   MINIMUM_CAPACITY * (\bigint)2 <= table.length  && // 4
  @   MAXIMUM_CAPACITY * (\bigint)2 >= table.length; // 2^29
  @
  @ public invariant // Non-empty keys are unique
  @   (\forall \bigint i; 0 <= i && i < table.length / (\bigint)2;
  @     (\forall \bigint j;
  @     i <= j && j < table.length / (\bigint)2;
  @     (table[2*i] != null && table[2*i] == table[2*j]) ==> i==j));
  @
  @ public invariant // Size == number of non-empty keys
  @   size == (\num_of \bigint i; 0 <= i < table.length / (\bigint)2;
  @                             table[2*i] != null);
  @
  @ public invariant // Table length is always an even number
  @   table.length % (\bigint)2 == 0;
  @
  @ // Table must have at least one empty key-element to prevent
  @ // infinite loops when a key is not present.
  @ public invariant
  @   (\exists \bigint i;  0 <= i < table.length / (\bigint)2;
  @     table[2*i] == null);
  @
  @ // There are no gaps between a key's hashed index and its actual
  @ // index (if the key is at a higher index than the hash code)
  @ public invariant
  @   (\forall \bigint i; 0 <= i < table.length / (\bigint)2;
  @     table[2*i] != null &&
  @     2*i > \dl_genHash(table[2*i], table.length) ==>
  @     (\forall \bigint j;
  @       \dl_genHash(table[2*i], table.length) / (\bigint)2 <= j < i;
  @       table[2*j] != null));
  @
  @ // There are no gaps between a key's hashed index and its actual
  @ // index (if the key is at a lower index than the hash code)
  @ public invariant
  @   (\forall \bigint i;  0 <= i < table.length / (\bigint)2;
  @     table[2*i] != null &&
  @     2*i < \dl_genHash(table[2*i], table.length) ==>
  @     (\forall \bigint j; \dl_genHash(table[2*i], table.length)
  @           <= 2*j < table.length || 0 <= 2*j < 2*i;
  @       table[2 * j] != null)); @*/
```

Listing 2. Excerpt of the class invariant.

of the iteration) may be encountered in this order during the execution of the loop:

- If $\texttt{hash} \leq \texttt{i} < \texttt{len} - 2$ then the updated value of $\texttt{i}$ is $\texttt{i} + 2$, so clearly the value of the variant has decreased from $\texttt{hash} + \texttt{len} - \texttt{i}$ to $\texttt{hash} + \texttt{len} - (\texttt{i} + 2)$ and remains non-negative (as $\texttt{hash} \geq 0$ and $\texttt{i} < \texttt{len} - 2$.)
- If $\texttt{i} = \texttt{len} - 2$ then the new value of $\texttt{i}$ is 0, so the variant decreases from $\texttt{hash} + \texttt{len} - (\texttt{len} - 2) = \texttt{hash} + 2$ to $\texttt{hash}$ (and $\texttt{hash} \geq 0$).
- If $0 \leq \texttt{i} < \texttt{hash} - 2$, the updated value of $\texttt{i}$ is $\texttt{i} + 2$ and the variant decreases from $\texttt{hash} - \texttt{i}$ to $\texttt{hash} - (\texttt{i} + 2)$ and so remains positive.
- If $\texttt{i} = \texttt{hash} - 2$ then the loop invariant implies that all slots for keys in the $\texttt{tab}$ array in the intervals $[0, \texttt{hash} - 2]$ and $[\texttt{hash}, \texttt{len} - 2]$ are not equal to k, the key that we searched for, and non-null (in other words, all keys except the one at $\texttt{i} = \texttt{hash} - 2$). If $\texttt{tab}[\texttt{hash} - 2] = k$ then clearly the return

```
/*@ // Index i is always an even value within the array bounds
  @ maintaining
  @   i >= 0 && i < len && i % (\bigint)2 == 0;
  @
  @ // Suppose i > hash. This can only be the case when no key k
  @ // and no null is present at an even index of tab in the
  @ // interval [hash..i-2].
  @ maintaining
  @   i > hash ==>
  @     (\forall \bigint n; hash <= (2*n) < i;
  @       tab[2*n] != k && tab[2*n] != null);
  @
  @ // Suppose i < hash. This can only be the case when no key k
  @ // and no null is present at an even index of tab in the
  @ // intervals [0..i-2] and [hash..len-2].
  @ maintaining
  @   i < hash ==>
  @     (\forall \bigint n; hash <= (2*n) < len;
  @       tab[2*n] != k && tab[2*n] != null) &&
  @     (\forall \bigint m; 0 <= (2*m) < i;
  @       tab[2*m] != k && tab[2*m] != null);
  @
  @ decreasing hash > i ? hash - i : hash + len - i;
  @ assignable \strictly_nothing; @*/
```

Listing 3. Loop specification of the loop in the `get` method and the inner loop of the `put` method.

statement on line 26 is hit. Otherwise, since the assignable clause states that the heap is not modified by the loop, we know the class invariant holds, which implies there must be an empty key slot in the array. This must be $\texttt{tab}[\texttt{hash} - 2] = \texttt{null}$ since all other key slots were non-null. In this case, the return statement on line 27 is hit and the loop terminates.

put(..) inner loop assignable clause. The assignable clause (Lst. 3) is peculiar: the code has an assignment to an array element (which is not dead code), yet the clause states that no locations are modified. This is due to the meaning of loop specifications: they must hold whenever the loop guard is checked. This however is not the case after leaving the loop by a return statement. Therefore in our case the assignable clause does not have to hold for the loop iteration in which the return statement is reached, and this is the case whenever the assignment that modifies the `table` is reached.

This strong assignable clause is very useful to prove the remainder of the method: all facts true before the loop (this may include the class invariant) are still valid and can be exploited after the inner loop!

put(..) satisfies contract and preserves class inv. We distinguish three scenarios with respect to the `put` method and wrote a contract for each of them. A so-called exceptional contract for the case that the hash map is full (it has reached max capacity): in that case the map is not modified and an exception is thrown. Another contract for the case that the map already contains the given key: then the corresponding value is updated. And a contract for the case where the table does not contain the given key yet so that the new key/value pair must be added. We shall focus on the proof of this last contract and discuss the main reasoning to show formally that, assuming the class invariant and precondition hold initially, **`put`** preserves the class invariant and satisfies the postconditions of this contract. This is the proof obligation that must be proven at line 41.

Consider the postcondition on line 10 of Lst. 4, about the preservation of old entries. The table is modified at `table`$[$`i`$]$ and `table`$[$`i` $+$ `1`$]$ which are **`null`**, as per the loop guard. So clearly, none of the entries that were already present are overwritten. In particular, in the case where the `table` is not resized, the old entries are at exactly the same index as at the beginning of the method. If the table *was* resized, the postcondition in the contract of `resize` (not shown) guarantees that they are present. The second main postcondition on line 18 is easy to establish: it says that there exists an index in the new table at which the new entry is stored. At line 41 we know that `i` is that index.

Next, we focus on two of the class invariants. The invariants that there are no gaps (key indices with a **`null`**) between the hash of any key and its actual index in the table (lines 24 and 34) are satisfied for the new entry: this follows from the invariant of the inner loop in put, Lst. 3 lines 7 and 15. For old entries, these properties remain true, because the method only overwrites a **`null`** entry, so it does not introduce new gaps. Hence, if there previously was no gap between an old key's hash and its index, then certainly there is not one after inserting the new key either.

Finally, we discuss the invariant that the map maintains at least one empty spot in `table` (line 18). The main challenge here is that `table`$[$`i`$]$ was previously **`null`** (i.e. it was an empty spot) and is now overwritten with the key object, so is there guaranteed to be an empty spot *elsewhere*? Note that the capacity of the table, i.e. the number of entries that can be stored, is `len`$/2$ since every entry (key and value) occupies two indices. If the old size is smaller than `len`$/2 - 1$, where `len` is the new length of the `table`, we can establish the desired property from the previous class invariant: as the size is the number of non-null entries (line 11) there must have been at least two empty spots. We now show that the old size is indeed smaller than `len` $- 1$ whenever we reach the **`return`**-statement on line 41. The if-statement prior to it must then have been false (otherwise control jumps back to the beginning of the outer loop with the **`continue`** statement). Hence, one of the following two cases is true:

– If `s` $+$ $($`s` $\ll$ `1`$) >$ `len` (where `s` is the new `size`) then `resize` must return *false*. This happens when the table length was at the maximum capacity already (so `resize` does not allocate a new table; it is a no-op) and the current size is less than that capacity - 1. If the size is equal to the max capacity - 1, `resize` (and **`put`**) throw an exception so the table is not modified.

– Otherwise `s` $+$ $($`s` $\ll$ `1`$) >$ `len` is false. Simplifying the left shift to 2`s` yields 2`s` $+$ `s` $>$ `len`. If `s` ≤ 3, at most six array indices in the table are used, but the table length is at least eight (line 2, where `MINIMUM_CAPACITY` $= 4$). So there must be an empty spot. If `s` > 2 then 2`s` $+$ `s` $\leq$ `len` implies 2`s` $+ 2 <$ `len`.

Some arithmetic reasoning about inequalities then suffices to establish the desired $\mathtt{s} < \mathtt{len}/2 - 1$.

```
/*@ also private exceptional_behavior ...
  @ also ... // The key is already present in the table
  @ also public normal_behavior // The key is not present in the table
  @   requires size < MAXIMUM_CAPACITY - 1;
  @   requires !(\exists \bigint i; 0 <= i < table.length/(\bigint)2;
  @                               table[i*2] == maskNull(key));
  @   assignable size, table[*], modCount, table;
  @   ensures size == \old(size) + 1 && modCount != \old(modCount)
  @           && \result == null;
  @   ensures // After execution, all old keys are still present
  @           // and all old values are still present
  @     (\forall \bigint i;
  @         0 <= i < \old(table.length) / (\bigint)2;
  @         (\exists \bigint j;
  @             0 <= j < table.length / (\bigint)2;
  @             (\old(table[i*2]) == table[j*2]) &&
  @              \old(table[i*2+1]) == table[j*2+1]));
  @   ensures // After execution, the table contains the new key
  @           // associated with the new value
  @     (\exists \bigint i;
  @         0 <= i < table.length / (\bigint)2;
  @         table[i*2] == maskNull(key) && table[i*2+1] == value); @*/
public /*@ nullable @*/ Object put(/*@ nullable @*/ Object key,
                                   /*@ nullable @*/ Object value) {
  final Object k =  maskNull(key);
  retryAfterResize: for (;;) {
    final Object[] tab = table; final int len = tab.length;
    int i = hash(k, len);
    //@ ghost \bigint hash = i;
    /*@ // Loop invariant: see Listing 3 @*/
    for (Object item; (item = tab[i]) != null;
      i = nextKeyIndex(i, len)) {
      if (item == k) {
        java.lang.Object oldValue =  tab[i+1];
        tab[i+1] = value; return oldValue; } }
    final int s =  size + 1;
    // Use optimized form of 3*s. Next capacity is len, 2*capacity
    if (s + (s << 1) > len && resize(len))
      continue retryAfterResize;
    modCount++; tab[i] = k; tab[i + 1] = value;
    size = s; return null; } }
```

Listing 4. The put method, including specifications.

4.1 Mechanic proof

We specified 15 methods of the `IdentityHashMap` and verified in KeY that they satisfy their contracts and preserve the class invariant: the default constructor with accompanying `capacity` and `init` methods (responsible for establishing the class invariant initially), the observers `isEmpty`, `maskNull`, `nextKeyIndex`, `size`, `unmaskNull`, the lookup methods `containsKeY`, `containsMapping`, `containsValue`, `get` and mutators `clear`, `put` and the private `resize` method. Tab. 1 summarizes the main statistics. The observer methods all have short proofs ($<$ 1,000 steps) and no interactive steps. All lookup methods have similar statistics: around 50k steps per contract. KeY's support for user interaction was crucial and

Table 1. Lines of code, lines of specification, and KeY statistics per method

Method	Steps	Br.	IS	SE	QI	OC	LI	MR	PO	JML	LOC
Def.constructor	7,724	56	86	66	101	1	0	0	1	10	3
`clear`	17,588	78	0	115	79	0	1	0	1	19	7
`containsMapping`	55,611	146	8	484	458	6	1	0	1	17	14
`put`	973,404	4,088	1,655	2,221	1,564	26	4	2	3	70	24
`resize`	223,357	340	487	491	270	3	2	0	4	125	29
other	172,307	438	115	846	1,243	14	4	0	13	113	59
Totals	1,449,991	5,146	2,351	4,223	3,715	50	12	2	23	354	136

Br.: Number of branches in the proof tree, **IS**: Interactive Steps (number of interactively (manually) applied rules), **SE**: Symbolic Execution steps, **QI**: Quantifier Instantiations, **OC**: Operation Contract applications, **LI**: Loop Invariant applications, **MR**: Merge Rule applications, **PO**: Proof Obligations (contracts) for the method, **JML**: lines of JML spec. (KeY only, not counting empty and comment lines), **LOC**: Lines Of Code (Java code not counting empty and comment lines).

used extensively to introduce intermediate lemmas and find suitable quantifier instantiations in the proofs of the most complex methods: `put` and `resize`.

The `IdentityHashMap` uses features for performance that complicate reasoning, such as `continue` jumps in loops, bit shifts and exploiting integer overflows. To match the intricate Java semantics, we took special care to analyze the source code nearly verbatim. We stripped generics with an automated plug-in of the KeY tool suite. The total effort of the case study amounts to roughly five person months (800 hours). The largest part of this consists of developing the formal specifications. This required many iterations of partial (failed) verification attempts with KeY and other analysis techniques (see Sect. 5.1) that led to corrections or additions to the specifications. With complete specifications, we estimate that the KeY proofs alone can be done in about 80 hours.

The `put` method, together with the private method `resize` was the largest and most difficult, comprising about 1.2 million steps together. The size is caused mainly because the class invariant is large and must be proven in every proof branch of a `return` statement. To minimize the number of such branches, we aggressively used a branch merging technique [17]. For example, line 41 of `put` gives rise to three branches: $\mathtt{s} + (\mathtt{s} \ll 1) > \mathtt{len}$ is false (branch 1), or it is true but `resize` returns false (branch 2) or true (branch 3). In branch 1 and branch 2 the heap is not modified, so we merged these branches. This prevents, for example, having to proving the class invariant twice.

Another valuable feature in KeY for `put` was the flexibility to verify loops by either unrolling the loop (with symbolic execution) or by supplying a loop invariant on a case-by-case basis. Observe that the body of the outer loop (line 26 is executed either just once (in case no resize is necessary) or twice (in case of a resize). To avoid having to write and use a (complex) loop invariant that complicates the proof, we exploited the feature of KeY to unroll the loop body instead. This is why there is no invariant for the outer loop.

5 Engineering Specifications Using Lightweight Analyses

Most of the time in a modular verification endeavor is spent on finding appropriate specifications, and we need to distinguish between two types of specification: While *property specifications* describe the exported guarantees one wants to verify, *auxiliary specifications* (like loop-invariants and contracts of helper methods) partition the verification condition into smaller obligations and guide tools to find proofs. In the present case study, coming up with both categories had challenges in store. To gain more confidence in the specifications and spot bugs early in the process, we applied two lightweight verification techniques.

5.1 Bounded Analysis for Auxiliary Specifications

Coming up with appropriate auxiliary specifications is a challenging task, because the specifications usually depend on each other in two directions: In modular verification, it is not possible to prove a method contract containing a method call without a specification of the called method. On the other hand, the inner method is difficult to specify while it is not clear what guarantees are needed at its call sites. It is thus very desirable to reduce these interdependencies and to step back from the design-by-contract paradigm for the inner method call. We achieved this by using a bounded analysis to check partially specified programs.

We use JJBMC [3] with which modular and bounded verification techniques can be combined: methods (and loops) with specifications are treated modularly (exploiting user-given method contracts and loop invariants to abstract from the program flow) while unspecified constructs can be formally treated using bounded verification (performing loop unrolling and method inlining to obtain a finite program to analyze), enabling a formal (albeit bounded) analysis of partially specified programs. The bounded analysis is parameterized by the maximum number $k \in \mathbb{N}$ of unwindings and unrollings to apply. For a too small value of k, specification violations may hence remain undiscovered by a bounded analysis.

The workflow to engineer auxiliary specifications is as follows: The user annotates a top-level API method m_0 with the desired property specification together with candidate class invariants (but leaves inner methods unspecified). They then run JJBMC to get feedback whether this specification is correct (within the set bound). If it is not, a concrete counterexample trace is produced and presented to the user who can use it for debugging. Once a suitable specification has been found, the user can continue engineering the specification for a method m_1 called by m_0. By continuously checking the bounded correctness of m_0 and the modular correctness of m_0 (wrt. the contract for m_1), the user hones in on an appropriate specification (strong enough for the call sites and weak enough to be provable) for m_1. The process then continues with the next nested method call, and also applies to (nested) loops. Using the bounded model checking analysis, we gained confidence in the specifications and avoided a few tedious refactorings otherwise needed for the proofs of the unbounded case.

As one example where this process helped us in the case study, reconsider the specification of `get` in Lst. 1. In the first specification attempt, the conditions in line 7 missed the call to `maskNull`, making code and specification inconsistent. Using JJBMC we were able to spot and correct this flaw early on before the inner mechanisms of `get` had been looked at. We used this approach to come up with several parts of the specification, and while we do not have hard evidence, our subjective impression is that it allowed us to get to correct specifications faster than we would have without it. We spent about 0.14 person months to verify the `IdentityHashMap` with JJBMC.

5.2 Unit Tests for Property Specifications

Dynamic techniques that check whether specifications hold at run-time could be cheap to apply, provided those checks are generated automatically from the JML specifications. There are tools designed for this purpose: JMLUnitNG [18] aims to generate unit tests and OpenJML [6] is a general analysis framework that includes support for run-time assertion checking. However, our application of these tools to this case study was unsuccessful: the semantics of the source code and specifications proved to be too complex and intricate to load the `Identity-HashMap`. In particular, this triggered exceptions and we did not manage to get useful output of the tools (despite contacting the main developer of OpenJML).

Confronted with this problem, we instead manually wrote (ad-hoc) JUnit tests to perform checks on method contracts (both pre- and postconditions) and a test method for the class invariant that checks all clauses. We can then call the test method whenever the class invariant should hold. Since the class invariant accesses private fields such as `table`, we used Java Reflection (`Class.getDeclaredField(..)`) to read the values of these fields. We handled quantifiers in JML specifications with for-loops (all quantifiers are bounded over the integers in our case study, so they can be translated routinely to for-loops).

Conducting these tests helped us to gain confidence in our specifications and even uncovered some errors in early versions of it. However, there are two main limitations: first, since JUnit tests operate at the granularity of entire methods, internal specifications such as loop invariants and assignable clauses are difficult to cover. Secondly, the manual translation of the JML specifications could be inconsistent (e.g. due to a misunderstanding of the semantics of JML) with the actual specification. Finally, as we use unit tests to discover errors quickly, one should keep in mind that writing and maintaining the unit tests is very time-consuming. We spent about 0.5 person months to develop the unit test framework.

6 Discussion

6.1 Empirical Identification of Verification Challenges

To learn more about the particular challenges imposed by the verification of hash tables, we not only verified the `IdentityHashMap`, but also investigated the contributing factors for the complexity of hash table verification endeavors in

Table 2. Required number of rules applications for different hash table implementations. The dash "–" denotes a non-closed proof.

	Separate Chaining			Linear Probing		
Method	WI	NE	WE	WI	NE	WE
constructor	24,096	*0.90*	–	3,577	*1.06*	*1.04*
`get`	15,353	*1.26*	*1.02*	1,160	*1.35*	*3.32*
`put`	82,624	*2.72*	–	29,632	*1.63*	–
`delete`	32,060	*1.44*	–	15,290	*1.68*	–
`hash`	1,303	*1.10*	*2.80*	1,061	*1.37*	*7.03*
`getIndex`	3,460	*0.93*	*6.65*	44,216	*1.61*	*5.70*
`addNewPair`	58,964	–	–	385,191	–	–
total	217,860			480,127		

KeY. We considered two families of hash table implementations with different hashing paradigms. For each family, we provided three implementations with different complexity and abstraction levels and compared the effort needed to verify them using KeY. To make the results more comparable and not influenced by user input, we have run KeY fully automatically without user interaction. The specification has a similar degree of abstraction and follows similar lines as the one outlined in Sect. 4. By comparing the required number of proof steps for the different implementations we can draw conclusions about the complexity of the verification obligations and the strengths and weaknesses of the automated proof strategy in KeY.

The two compared hash collision resolution paradigms are *linear probing* and *separate chaining*. They differ in situations in which two different keys map to the same hash (index) into the hash map. Linear probing is used in the `IdentityHashMap`, as described in Sect. 3. Separate chaining is used in the `HashMap` class in the JDK. It allows storing multiple entries into one slot: each slot contains a bucket (i.e. a linear list) with all entries that are mapped by the hash function to the same index (slot). The collision resolution strategy affects the algorithms for insertion and lookup routines since these have to take conflicting keys with identical indices into account. The implementations of the two paradigms have quite different method contracts and in particular the class invariants capturing the properties of the hash structure differ considerably – with different challenges both for the specifier and the automatic verification engine.

The three variants implemented for each conflict resolution strategy mainly differ in the data types used for values and keys. In the first variant called WithIntegers (WI), keys and values are of type `int` and the identity operator (==) is used to compare keys. The second variant is called NoEquals (NE), and keys are objects of a specialized immutable `Key` class, while values are arbitrary `Object`s, and it uses the identity operator (==) to compare keys, like the `IdentityHashMap` does. The third variant is called WithEquals (WE) and is similar to NE, but uses the `equals` method to compare keys.

Tab. 2 shows the required effort to prove the respective method contracts correct. The numbers of the WI variants represent the absolute number of rule applications needed, whereas the other two variants (in italics) are stated as a ratio to the number in the WI column for the same method and hashing family. Thus the relative overhead between the family members can be seen more easily. The exact numbers of steps are not very important for the investigation, suffice it to say that the WI proof for addNewPair with more than 380,000 steps took 12 minutes to complete. The `hash` method computes the hash value of a key and `getIndex` returns the index of a key (if present). `addNewPair` inserts a new key-value pair and is called by `put`, when the key is not already in the hash table. Some proofs could not be finished (indicated as −) since the prover ran out of memory resources. Since the solver is designed to be deterministic, the runs are repeatable.

It can be observed that in most cases the complexity grows within a family between the variants from WI to NE and from NE to WE. The variants that introduce `equals` instead of `==` experience a vast increase in complexity for the central method `getIndex`. This can be explained by the fact that the built-in identity comparison is independent of the heap state (it only depends on the compared values) and is inherently transitive and symmetric. Such properties may (or may partially) be true for an `equals` implementation, but considerably more effort must be taken to show consequences when dealing with this more general form of equality. It can thus be safely said that one should use primitive values as keys for hash maps as often as possible for KeY.

Contrary to what one might expect, some numbers decrease for the more complex variants. This can be explained by the heuristic choices that are made by the KeY strategy. In some cases, good decisions are made earlier than in other cases, due to the presence/absence of certain trigger expressions.

6.2 Discovered Bugs and Recommendations

In this section, we discuss several issues that our analysis revealed.

Serialization. The `IdentityHashMap` supports serialization: writing a map to a stream (e.g. a file) with a `writeObject` method and reading a map from a stream with the `readObject` method. Effectively `readObject` acts as a constructor: it creates a map object, so it should ensure that this object satisfies the class invariant. To fill the map with serialized entries, `readObject` uses a `putForCreate` method that does not resize (for performance reasons) but allocates a table based on the size stored in the stream. Suppose an attacker serializes a map with a single empty entry (satisfying the requirement from the class invariant that there is an empty slot) to a file. The attacker can tamper with the file using a hex-editor to overwrite the empty slot with a key. A victim who deserializes this rogue map then inadvertently enters an infinite loop in `putForCreate`. We suggest solving this by checking in the code whether the map to deserialize satisfies the class invariant, and if not, throw an exception to prevent infinite loops or construction of a map object that breaks the class invariant.

***put** in JDK7u80.* The binaries distributed by Oracle for JDK7 (an older but still widely used JDK) uses source code from an old JDK7u80 update[6]. The main difference between JDK7u80 and the **IdentityHashMap** in this paper (which is used in all newer JDK's and the later source-only updates to JDK7) is in the **put** method. The JDK7u80 version resizes *after* adding a new entry, rather than before (see Fig. 3), and there is no outer loop with a **continue** statement. Suppose **put** is called on a map that is filled to the maximum capacity. The last empty spot in the table is first overwritten and only then **resize** throws an exception. So, the map is left in an inconsistent state: it breaks the class invariant. If a client then calls **get(k)** on a key **k** not stored in the map, an infinite loop is triggered. In other words: this version of **put** breaks *failure atomicity*: put fails (as the table is full) so the operation should have been a no-op.

```
... // See Lst. 4, lines 27 − 35
tab[i] = k; tab[i + 1] = value;
if (++size >= threshold)
  resize(len);
  // len == 2 * current capacity.
return null;
```

Fig. 3. put(..) in JDK7u80

There is a way to fix this without resorting to **continue** statements: extract the code for the inner loop in **put**, **get**, etc., which searches for the index of a given key, or returns the index of its insertion point if the key is not present in the map, into a new private method **search(k)**. The duplicated code for the loop can then be eliminated from the various methods by calling **search**. In **put**, call resize before modifying the table. This may shuffle around the existing keys: the hashes are recalculated based on the new table length. If a resize occurred, call **search(k)** again to obtain the new insertion point for the key. Now the entries can be safely inserted at the index returned by **search**.

7 Conclusion

In this paper we specified and verified the core of the challenging, real-world implementation **IdentityHashMap** in KeY and discovered several issues. To speed up finding suitable specifications, we successfully leveraged model checking and unit testing. We extended our analysis with an investigation on the effect on the proof complexity in KeY of features and strategies used in other map implementations.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6

[6] http://hg.openjdk.java.net/jdk7u/jdk7u-dev/jdk/file/70e3553d9d6e/src/share/classes/java/util/IdentityHashMap.java

2. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (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. Lecture Notes in Computer Science, vol. 12476, pp. 60–80. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_4
3. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. Springer (2020)
4. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving jdk's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10712, pp. 35–48. Springer (2017). https://doi.org/10.1007/978-3-319-72308-2_3, https://doi.org/10.1007/978-3-319-72308-2_3
5. de Boer, M., de Gow, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Artifacts of the Formal Specification and Verification of JDK's Identity Hash Map Implementation (Mar 2022). https://doi.org/10.5281/zenodo.6415339, https://doi.org/10.5281/zenodo.6415339
6. Cok, D.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. Electronic Proceedings in Theoretical Computer Science **149** (Apr 2014). https://doi.org/10.4204/EPTCS.149.8
7. Cok, D.R.: Openjml: JML for java 7 by extending openjdk. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_35
8. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying java bytecode. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 183–190. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
9. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.J.: Behavioral interface specification languages. ACM Comput. Surv. **44**(3), 16:1–16:58 (2012). https://doi.org/10.1145/2187671.2187678
10. Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: History-based specification and verification of java collections in key. In: Dongol, B., Troubitsyna, E. (eds.) Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12546, pp. 199–217. Springer (2020). https://doi.org/10.1007/978-3-030-63461-2_11
11. Knüppel, A., Thüm, T., Pardylla, C., Schaefer, I.: Experience report on formally verifying parts of OpenJDK's API with KeY. In: F-IDE 2018: Formal Integrated Development Environment. EPTCS, vol. 284, pp. 53–70. OPA (2018). https://doi.org/10.4204/EPTCS.284.5
12. Knuth, D.E.: Notes on "open" addressing (July 1963), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4899
13. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., et al.: JML reference manual (2008)

14. Meyer, B.: Applying ”design by contract”. Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
15. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: FM 2015: Formal Methods. LNCS, vol. 9109, pp. 414–434. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_26
16. Pottier, F.: Verifying a hash table and its iterators in higher-order separation logic. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017. pp. 3–16. ACM (2017). https://doi.org/10.1145/3018610.3018624, `https://doi.org/10.1145/3018610.3018624`
17. Scheurer, D., Hähnle, R., Bubel, R.: A general lattice model for merging symbolic execution branches. In: Ogata, K., Lawford, M., Liu, S. (eds.) Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10009, pp. 57–73 (2016)
18. Zimmerman, D.M., Nagmoti, R.: Jmlunit: The next generation. In: Beckert, B., Marché, C. (eds.) Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6528, pp. 183–197. Springer (2010). https://doi.org/10.1007/978-3-642-18070-5_13, `https://doi.org/10.1007/978-3-642-18070-5_13`