The Binomial Heap Verification Challenge in Viper

Peter Müller

Abstract Binomial heaps have interesting invariants that constrain the shape of and the values stored in the data structure. A challenge of the VSComp 2014 verification competition was to find a fault in a given Java implementation of binomial heaps that leads to a violation of the invariants, to fix the error, and to verify the corrected version. In this paper, we present the first solution to this challenge. Using an encoding of the verification problem into Viper, we identified and fixed the known and a previously-unknown fault in the Java code and then successfully verified the implementation. Our case study illustrates the degree of automation that modern program verifiers achieve for complex invariants; it also demonstrates how modular verification techniques can be used to iteratively strengthen the verified properties, allowing the developer to focus on one concern at a time.

1 Introduction

The program verification community regularly engages in verification competitions such as SV-COMP, VerifyThis, and VSComp. The participants attempt to solve a set of challenges using verification tools of their choice. These competitions allow tool developers to assess the strengths and weaknesses of their tools and identify techniques and tool features that are useful to handle challenging verification problems. It is common for participants to work on their solutions even beyond the end of the official competition, and to publish their results [11].

In this paper, we present our solution to one of the challenges of the VSComp 2014 competition. The challenge is to detect a fault in the Java

Peter Müller
Department of Computer Science, ETH Zurich, Switzerland, e-mail: peter.mueller@inf.ethz.ch

implementation of a binomial heap data structure [4], to fix the error, and to verify that the corrected code maintains the invariants of binomial heaps. VSComp 2014 was organized by Ernie Cohen, Marcelo Frias, Natarajan Shankar, and the author of this paper. The binomial heap challenge was proposed by Marcelo Frias. Out of 14 teams that had registered for the competition, 8 submitted solutions to some of the challenges. None of them solved the binomial heaps problem; to the best of our knowledge, it has also not been solved in the aftermath of the competition.

We solved this challenge using our verification infrastructure Viper [16]. We translated the Java program into the Viper intermediate verification language, annotated the Viper program with suitable specifications, and verified it using Viper's symbolic-execution-based verification back-end. The solution uses access permissions and recursive predicates in the style of separation logic to describe the shape of the binomial heap data structure, and heap-dependent abstraction functions to specify value invariants such as sortedness of lists. The method specifications and loop invariants necessary for the verification include a mix of complex shape and value properties. Our case study illustrates the degree of automation that modern program verifiers achieve for such properties; it also demonstrates how modular verification techniques can be used to iteratively strengthen the verified properties, thereby allowing the developer to focus on one concern at a time.

A noteworthy outcome of our verification effort is that we did not only find and fix the defect that was detected earlier by Marcelo Frias using bounded model checking. We also identified a second (very similar) fault that was previously undetected. Both faults are present in Java implementations of binomial heaps that are available online, for instance, at www.sanfoundry.com/java-program-implement-binomial-heap.

Outline. We provide the necessary background on binomial heaps in Sect. 2. We reproduce the challenge in Sect. 3 and explain the intended behavior of the faulty method in Sect. 4. We summarize our formalization of the main invariants in Sect. 5 and explain in Sect. 6 how we found the fault, fixed it, and verified the corrected code. Sect. 7 provides a quantitative and qualitative evaluation of our solutions and Sect. 8 concludes. The challenge statement including the given Java source code as well as our solution are available online [14].

2 Binomial Heaps

A *binomial heap* stores a multiset of keys, here, integers. Important operations are very efficient: finding a minimum value, deleting a minimum value, decreasing a key, and merging two binomial heaps all work in logarithmic amortized time; insertion has constant amortized time.

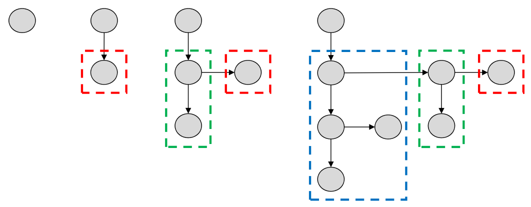


Fig. 1: Binomial trees of degrees 0, 1, 2, and 3. The children of a node are stored in a singly-linked list. Horizontal arrows depict the `sibling`-references between the children of a node. Vertical arrows depict `child`-references from a parent to its first child. We omit `parent`-references in the diagrams for simplicity. The dashed boxes visualize the structure. For instance, the children of a tree of degree 3 are three trees of degrees 2, 1, and 0.

Binomial heaps are sets of binomial trees. A *binomial tree* is defined as follows: a binomial tree of degree 0 is a single node. A binomial tree of degree k has k children, which are the roots of binomial trees of degrees $k-1, k-2, \ldots, 1, 0$. The list of children is ordered in descending order. Consequently, a binomial tree of degree k has 2^k nodes. Fig. 1 shows binomial trees.

A binomial heap contains at most one binomial tree of each degree. Each tree must satisfy the minimum-heap property, that is, the key of each node is less or equal to the keys of each of their children. Consequently, the minimum key of each tree is stored in its root. Fig. 4a shows a binomial heap with trees of degrees 0, 2, and 3.

3 Verification Challenge

The verification challenge addressed in this paper was presented at VS-Comp 2014 as follows.

> We present a compilable excerpt from classes `BinomialHeap` and `BinomialHeapNode` modeling binomial heaps [4]. Method extractMin has a fault that allows for the class invariant to be violated. Your goals towards solving this problem are:
>
> 1. Find and describe the above-mentioned fault.
> 2. Provide an input binomial heap that exercises this fault, and describe the technique used to find the input (automated techniques are preferred to human inspection).
> 3. Provide a correct version of method `extractMin`.
> 4. Provide a suitable class invariant.
> 5. Verify that method `extractMin` indeed preserves the provided class invariant or at least a meaningful subset of properties from the class invariant.
>
> Hints can be obtained from the organizers in exchange for penalties in the final score.

```java
public class BinomialHeapNode {
  public int key;
  public int degree;
  public BinomialHeapNode parent;
  public BinomialHeapNode sibling;
  public BinomialHeapNode child;

  public BinomialHeapNode() {}

  public BinomialHeapNode reverse(BinomialHeapNode sibl) {
    BinomialHeapNode ret;
    if (sibling != null)
      ret = sibling.reverse(this);
    else
      ret = this;
    sibling = sibl;
    return ret;
  }
}
```

Fig. 2: The provided Java implementation of class `BinomialHeapNode`.

The given Java implementation of tree nodes, class `BinomialHeapNode`, is presented in Fig. 2. For binomial heaps, Fig. 3 presents an outline of class `BinomialHeap` including the method `extractMin`, which is mentioned in the challenge. We provide the complete source code from the competition problem online [14].

4 Algorithm

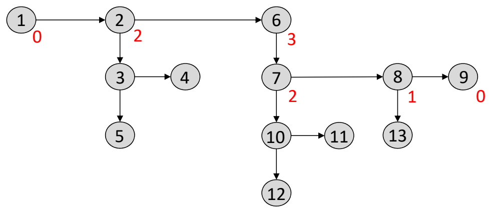
The challenge states that method `extractMin` has a fault. In this section, we explain the intended behavior of the method on a concrete example. The method is supposed to find a minimum key in a given binomial heap and remove it. It achieves that in five main steps, which we explain in the following and illustrate in Fig. 4.

Fig. 4a shows the input heap; we assume that node 6 contains the minimum key to be removed. Note that the implementation used in the challenge requires the trees in a binomial heap to be sorted by degree in ascending order.

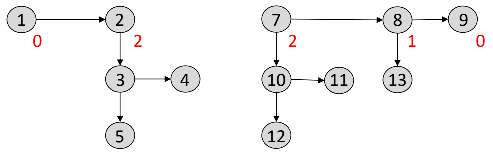
Step 1: Removing the Minimum Node. Method `extractMin` traverses the list of binomial trees and identifies the tree with the minimum key (via the call to `findMinNode` in line 9 of Fig. 3). Due to the minimum-heap property, this method needs to compare only the keys at the roots. Lines 10–16 of `extractMin` remove the tree with the minimum from the binomial heap. The remaining elements of the original binomial heap are then split between the

```java
public class BinomialHeap {
  public BinomialHeapNode Nodes;
  public int size;

  public BinomialHeapNode extractMin() {
    if (Nodes == null)  return null;

    BinomialHeapNode temp = Nodes, prevTemp = null;
    BinomialHeapNode minNode = findMinNode(Nodes);
    while (temp.key != minNode.key) {
      prevTemp = temp;
      temp = temp.sibling;
    }

    if (prevTemp == null) { Nodes = temp.sibling; }
    else                  { prevTemp.sibling = temp.sibling; }
    temp = temp.child;
    BinomialHeapNode fakeNode = temp;

    // remove the parent-pointers pointing to the minimum
    while (temp != null) {
      temp.parent = null;
      temp = temp.sibling;
    }

    if ((Nodes == null) && (fakeNode == null)) {
      size = 0;
    } else {
      if ((Nodes == null) && (fakeNode != null)) {
        Nodes = fakeNode.reverse(null);
        size--;
      } else {
        if ((Nodes != null) && (fakeNode == null)) {
          size--;
        } else {
          unionNodes(fakeNode.reverse(null));
          size--;
        }
      }
    }
    return minNode;
  }

  void merge(BinomialHeapNode binHeap) { ... }
  void unionNodes(BinomialHeapNode binHeap) { ... }
  static BinomialHeapNode findMinNode(BinomialHeapNode arg) { ... }
}
```

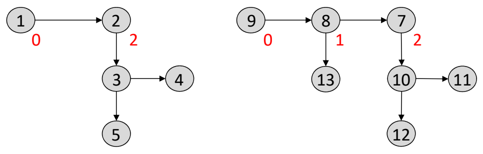
Fig. 3: Excerpt from the provided Java implementation of class `BinomialHeap`. The complete version is available online [14].



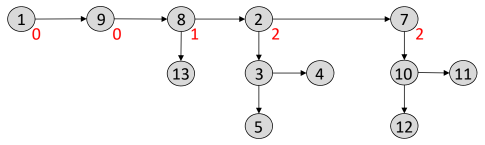
(a) The initial binomial heap.



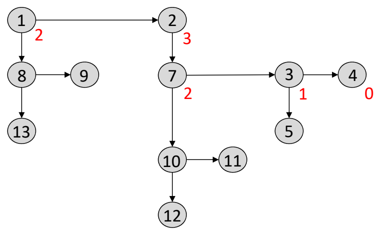
(b) State after removing the minimum node 6.



(c) State after reversing the children list of the removed minimum node.



(d) State after merging the two lists.



(e) Final binomial heap.

Fig. 4: Example execution of method `extractMin`. For illustration purposes, each node has a unique number; for some binomial trees, we indicate their degree in red. We assume that the minimum value to be extracted is stored in node 6. We show the states before and after each of the four steps of the algorithm. The diagram shows the result of a correct implementation, not the code provided in the challenge.

remainder of the heap and the children of the removed minimum node, as illustrated in Fig. 4b.

Step 2: Reversing the Children List. The children list of the removed minimum node is itself a binomial heap, but sorted in descending rather than ascending order. The next step of `extractMin` (lines 30 and 36 in Fig. 3) is to reverse this list, resulting in the state shown in Fig. 4c.

Step 3: Merging the Heaps. The next step (call to `unionNodes` in line 36) merges both binomial heaps into one sorted list of trees as shown in Fig. 4d. This merge operation is implemented by a method `merge`, which is called at the beginning of method `unionNodes`. Note that the resulting list is not a binomial heap since it may contain up to two trees for each degree. In our example, it includes two trees of degree 0 and two of degree 2.

Step 4: Combining the Trees. To re-establish the invariant of a binomial heap, method `unionNodes` traverses the list resulting from `merge` and iteratively combines two consecutive trees of degree k into one tree of degree $k+1$. This traversal relies on the fact that the trees of a binomial heap (and consequently the list produced by merging two heaps) are sorted in ascending order. If that is the case, it produces a well-formed binomial heap as shown in Fig. 4e. Here, the two trees of degree 0 were merged into a tree of degree 1, which is then merged with the other tree of degree 1 into a tree of degree 2, resulting in a list with three trees of degree 2. The two trees at the back of the list are then merged into one tree of degree 3.

5 Formalization of Invariants

We solved the verification challenge by manually encoding the given Java implementation and its specification into the intermediate verification language Viper [16]. Viper uses a program logic based on implicit dynamic frames [23], a permission logic akin to separation logic [17, 21]. Viper associates an *access permission* with each heap location. A method may read or write the location only if it holds the corresponding permission. Permissions may be passed between method executions, but cannot be duplicated or forged. Permissions to an unbounded number of memory locations can be represented by recursive predicates [18]. Viper's permission logic offers various advanced features such as fractional permissions [2, 9], magic wands [22], and quantified permissions (iterated separating conjunction) [15], but these features were not needed for the binomial heap challenge.

In contrast to separation logic, implicit dynamic frames separate permission specifications from value properties. For instance, separation logic's points-to predicate $x.f \mapsto y$ is expressed in Viper as the conjunction of two assertions `acc(`x`.`f`) && `x`.`f` == `y, where the former conjunct denotes the access permission to location $x.f$ and the latter constrains the value stored in the location. This separation carries over to other specification constructs. In particular, Viper uses recursive predicates to abstract over the permissions to a data structure and supports heap-dependent abstraction functions to abstract over its values [8]. Viper's conjunction `&&` behaves like separating conjunction when applied to assertions that denote permissions. In particular, `acc(`x`.`f`) && acc(`y`.`f`)` implies $x \neq y$.

```
 1  predicate tree(this: Ref) {
 2    acc(this.key) && acc(this.degree) &&
 3    acc(this.child) && acc(this.parent) &&
 4    0 <= this.degree &&
 5    heapseg(this.child, null) &&
 6    this.degree == segLength(this.child, null) &&
 7    (0 < this.degree  ==>
 8               segDegree(this.child, null, 0) == this.degree - 1) &&
 9    validChildren(this.child, null)  &&
10    (this.child != null ==> segParent(this.child, null) == this)
11  }
12
13  predicate heapseg(this: Ref, last: Ref) {
14    this != last ==>
15      tree(this) && acc(this.sibling) &&
16      heapseg(this.sibling, last) &&
17      (this.sibling != last ==>
18                   treeParent(this) == segParent(this.sibling, last))
19  }
```

Fig. 5: Mutually recursive predicates describing binomial trees.

We formalize the invariants of binomial trees using two mutually recursive predicates `tree` and `heapseg`, which are presented in Fig. 5. The `tree` predicate takes as argument a reference to a tree node; it provides access permission to the four fields of this node. The subsequent conjuncts express that a tree's degree is non-negative (line 4), that the `child` field points to a null-terminated list (line 5), and that the degree is the length of this list (line 6). Here, lists are represented as so-called list segments, encoded via the `heapseg` predicate described below. `segLength` is a function that yields the length of a list segment; the definition is straightforward and, therefore, omitted. The remaining conjuncts express properties of the children list: the first child's degree is one less than the current node's (lines 7 and 8); the boolean function `validChildren` (line 9) encodes that each child's degree is one larger than its sibling's. Finally, the first child's parent is the current node (line 10).

The `heapseg` predicate represents the empty list segment when its arguments are equal. Otherwise, it provides a `tree` predicate for each node in the list segment and access permission to its `sibling` field (line 15), and requires all nodes in the segment to have the same parent (lines 17 and 18).

Together, these predicates encode the invariant of binomial trees explained in Sect. 2: The access permissions ensure that the data structure is a tree since they prevent aliasing. Moreover, since the first child of a tree with degree k has degree $k-1$ (lines 7 and 8), and since there are k children (line 6), which are binomial trees (line 15) whose degrees decrease by one from child to child (function `validChildren` in line 9), we express that the children are the roots of binomial trees of degrees $k-1, k-2, \ldots, 1, 0$, as required.

```
1  predicate heap(this: Ref) {
2    acc(this.Nodes) &&
3    heapseg(this.Nodes, null) && sorted(this.Nodes, null) &&
4    (this.Nodes != null ==> segParent(this.Nodes, null) == null) &&
5    acc(this.size) && this.size == segSize(this.Nodes, null)
6  }
```

Fig. 6: Predicate describing binomial heaps.

The `heap` predicate in Fig. 6 describes the invariant of a binomial heap. It provides access permission to the `Nodes` field, which points to the root of the first binomial tree (line 2). The trees in a binomial heap form a null-terminated list; function `sorted` expresses that the trees in this list are sorted by degrees in strictly increasing order (line 3). The trees in this list have no parent (line 4). Finally, the code provided in the verification challenge has a `size` field that is supposed to store the number of elements in a binomial heap. This field is not read in the provided Java implementation. We included it, nevertheless, together with an invariant that `size` contains the actual number of elements as determined by the recursive function `segSize` (line 5).

6 Verification

The verification challenge (see Sect. 3) states that method `extractMin` violates the invariant of a binomial heap. We can check this property by verifying the following method specification in Viper: If the receiver object `this` is a well-formed binomial heap in the pre-state of the method (that is, satisfies the `heap` predicate), it will also be a well-formed heap in the post-state:

```
method extractMin(this: Ref) returns (res: Ref)
  requires heap(this)
  ensures  heap(this)
```

Verifying the method implementation against this specification requires suitable specifications for all methods (transitively) called by `extractMin`. These specifications are available online [14]. Here, we focus on method `merge`, which is the method that contains the fault. As part of Step 3 of the `extractMin` algorithm (see Sect. 4), `merge` merges two sorted lists of binomial trees as illustrated by the transition from Fig. 4c to Fig. 4d.

```
1  method merge(this: Ref, binHeap: Ref)
2   requires acc(this.Nodes) && this.Nodes != null
3   requires heapseg(this.Nodes, null) && sorted(this.Nodes, null)
4   requires heapseg(binHeap, null) && sorted(binHeap, null)
5   requires binHeap != null ==>
6            segParent(this.Nodes, null) == segParent(binHeap, null)
7
8   ensures  acc(this.Nodes) && this.Nodes != null
9   ensures  heapseg(this.Nodes, null) && presorted(this.Nodes, null)
10  ensures  segSize(this.Nodes, null) ==
11           old(segSize(this.Nodes, null))+old(segSize(binHeap, null))
12  ensures  segParent(this.Nodes, null)==old(segParent(this.Nodes, null))
```

Fig. 7: Viper specification of method `merge`.

The Viper specification of method `merge` is shown in Fig. 7. The first two preconditions require the receiver to be a non-empty binomial heap (lines 2 and 3). Viper distinguishes between a predicate and its body [24]. In order to prevent automatic provers from unrolling recursive definitions indefinitely, exchanging a predicate for its body and vice versa is done manually via `unfold` and `fold` statements. The first two preconditions include the unfolded version of the `heap(this)` predicate. The third precondition requires parameters `binHeap` to point to a null-terminated list of binomial heaps (line 4), and the last precondition requires the trees in both lists to have the same parent (line 5).

Similarly, the first two postconditions ensure that after the method execution, the `Nodes` field of the receiver will point to a non-empty, null-terminated list of binomial trees. Note, however, that this list will in general not satisfy the `sorted` property. As we explained in Sect. 4 and illustrated in Fig. 4d, the resulting list is sorted, but, in contrast to a binomial heap, may contain up to two trees of each degree. We express these properties using the `presorted` function (lines 8 and 9). The third postcondition expresses that the size of the merged list is the sum of the sizes of the input lists (lines 10 and 11), and the last postcondition ensures that the parent of the trees in the lists is not modified by the method (line 12).

This specification illustrates one of the main benefits of implicit dynamic frames over traditional separation logic. By separating access permissions from value properties, one can conveniently describe different constraints over the same data structure. Our `heapseg` predicate describes the permissions to the trees in a list segment, whereas different sorting criteria can be specified separately using functions. In the specification of method `merge`, we use `sorted` to express sortedness without duplicates and `presorted` to allow degrees to occur at most twice. During the combination of binomial trees (Step 4 in Sect. 4), there may even be states where a list can contain up to one element up to three times. For instance, after combining the first two trees in Fig. 4d and then combining the result with the third tree, the list includes three trees of degree 2. Viper allows one to use the same `heapseg` predicate in all three situations and to combine it with different value constraints. In contrast, separation logic would require either a complex parameterization of the predicate or a mapping from list segments to a mathematical sequence of degrees that can then be constrained appropriately; however, reasoning about sequences in SMT solvers is often flaky.

The Java implementation of method `merge` is presented in Fig. 8; its Viper encoding is available online [14]. Verifying this method requires a complex loop invariant consisting of 26 lines of Viper code. Besides various access permissions and sortedness criteria, which we discussed above, it requires that the degree of the last tree in the list starting at `temp1` is at most the degree of the tree pointed to by `temp2`, unless `temp1` points to the first tree, `this.Nodes`. This constraint is violated by the implementation given in the verification challenge.

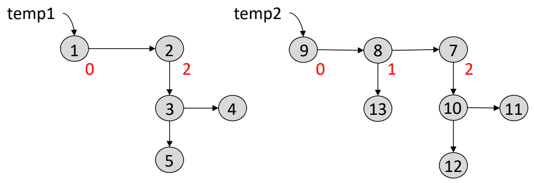
Fig. 9 shows the execution of the given faulty implementation of method `merge` for the example from Fig. 4. As shown by Fig. 9b, already the first loop iteration does not preserve this invariant: Here, the last tree in the listed starting at `temp1` has degree 2, which is larger than the degree of `temp2`, which is 1. A careful inspection of the executed branch of the loop (lines 5–9) reveals that the assignment `temp1 = tmp.sibling` in line 9 is wrong. The correct assignment is `temp1 = tmp`, which would let `temp1` point to node 9 in Fig. 9b and, thus, preserve the loop invariant. The smallest counterexample that reveals this error has 13 nodes in the initial binomial heap, as in Fig. 4a. We confirmed the detected fault also by running the Java implementation on that example.

Re-verifying the code after fixing this fault reveals a second one: the assignment in line 18 has the same fault and can be fixed in the same way. The smallest counterexample we found is a binomial input heap with 25 nodes: trees of degrees 0, 3, and 4, where the minimum is extracted from the tree of degree 3; we also confirmed this fault in the given Java implementation. After applying the second fix, the example verifies.

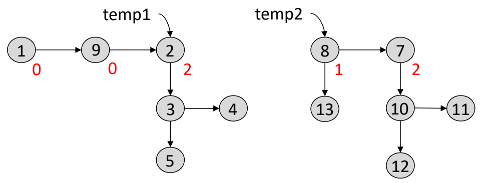
It is interesting to observe that the loop invariant implies that the last else-block (lines 23–29) is reachable only when `temp1 == this.Nodes`, a property that we verified. So the condition in line 27 is always true and the if-statement could be omitted.

```
private void merge(BinomialHeapNode binHeap) {
  BinomialHeapNode temp1 = Nodes, temp2 = binHeap;
  while ((temp1 != null) && (temp2 != null)) {
    if (temp1.degree == temp2.degree) {
      BinomialHeapNode tmp = temp2;
      temp2 = temp2.sibling;
      tmp.sibling = temp1.sibling;
      temp1.sibling = tmp;
      temp1 = tmp.sibling;
    } else {
      if (temp1.degree < temp2.degree) {
        if ((temp1.sibling == null)
            || (temp1.sibling.degree > temp2.degree)) {
          BinomialHeapNode tmp = temp2;
          temp2 = temp2.sibling;
          tmp.sibling = temp1.sibling;
          temp1.sibling = tmp;
          temp1 = tmp.sibling;
        } else {
          temp1 = temp1.sibling;
        }
      } else {
        BinomialHeapNode tmp = temp1;
        temp1 = temp2;
        temp2 = temp2.sibling;
        temp1.sibling = tmp;
        if (tmp == Nodes) {
          Nodes = temp1;
        }
      }
    }
  }

  if (temp1 == null) {
    temp1 = Nodes;
    while (temp1.sibling != null) {
      temp1 = temp1.sibling;
    }
    temp1.sibling = temp2;
  }
}
```

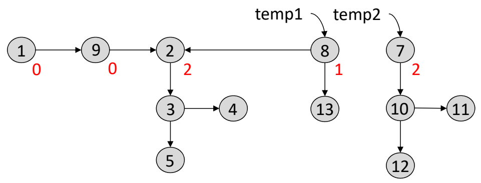
Fig. 8: Provided implementation of the auxiliary method `merge`, which is called from `unionNodes`, which is in turn called from `extractMin`.



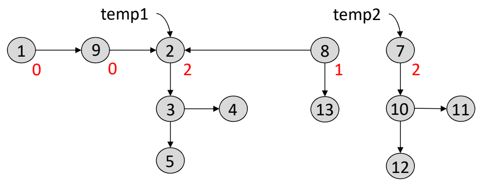
(a) State at the beginning of method `merge`.



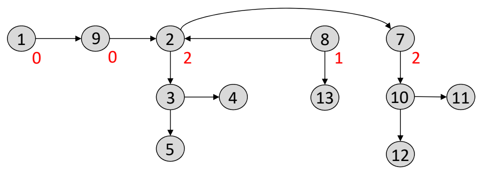
(b) State after the first loop iteration (executing lines 5–9).



(c) State after the second loop iteration (executing lines 23–29).



(d) State after the third loop iteration (executing line 20).



(e) State after the last loop iteration (executing lines 5–9).

Fig. 9: States before and after each loop iteration in the faulty implementation of method `merge`. The initial state is the same as in Fig. 4d. Fig. 4e shows the final state of the corrected implementation. The faulty implementation incorrectly removes nodes 8 and 13 from the list.

7 Discussion

We solved all five tasks given in the verification challenge (see Sect. 3). The input required for Task 2 was found manually when trying to understand why the loop invariant in method `merge` is not preserved. Note that our solution goes beyond the challenge in two major ways. First, Viper's permission logic ensures the absence of run-time errors and data races. That is, our solution is correct even in a concurrent setting. Second, before he proposed the verification challenge, Marcelo Frias used bounded model checking and detected the first of the two errors. However, the second error, which is triggered by a larger input (25 instead of 13 nodes) remained undetected. We found both errors and verified the fixed version of the code for unbounded inputs.

Table 1: Summary of solution. "Lines" excludes empty lines and comments.

Content	**Lines**
Program code	170
Predicate and function declarations	80
Method specifications for 5 methods	37
Ghost code including specification of ghost method	128
Loop invariants	112
Local assertions	13
Total overall	**540**
Total annotations	**370**

Quantitative Evaluation. Table 1 provides an overview of our encoding. The 141 lines of Java code were encoded into 170 lines of Viper code; the slight increase is caused by the fact that Viper is an intermediate language. The translation from Java to Viper could be performed completely automatically using a suitable front-end such as VerCors [1]. The specification of the program consists of 117 lines, 80 lines for the definition of recursive predicates and heap-dependent abstraction functions, and 37 lines for the preconditions and postconditions of the five methods. In addition, our solution required 128 lines of ghost code for the unfolding and folding of recursive predicates, 112 lines of loop invariants, and 13 local assertions to help the SMT solver, for instance, to unroll function definitions.

In total, the ratio of annotations to code is $370/170 = 2.2$, which is significantly lower than the numbers reported for other SMT-based verification efforts, which report an overhead factor around 5 [6, 7], and much lower than verification efforts in interactive theorem provers, which often have overhead factors between 10 and 20 [5, 10, 12, 26]. Note, however, that these are very rough comparisons since the complexity of programs and verified properties differ between these case studies; moreover, verification in interactive provers such as Coq can provide foundational guarantees, whereas soundness of our approach depends on the correct implementation of Viper.

We measured the verification time on an Intel Core i7-4770 CPU (3.40GHz, 16Gb RAM) running Windows 10 Pro using a warmed-up JVM. Averaged over ten runs, Viper's verification back-end based on symbolic execution requires slightly under 23 seconds for the verification.

Qualitative Evaluation. Viper's modular verification technique allowed us to verify the example incrementally. We started with the verification of memory safety, using recursive predicates that mostly specified access permission. We then iteratively strengthened the invariants, first by including more properties of binomial heaps and trees, and then by including the `parent` and `size` fields. Due to Viper's support for heap-dependent abstraction functions, we were able to strengthen the verified properties without substantial changes to the recursive predicates. This iterative process greatly reduced the complexity of the verification tasks.

During the verification, we simplified the code in three minor ways. First, we moved line 16 in method `reverse` (Fig. 2) into the conditional statement, which allowed us to fold the `heapseg` predicate before the recursive call in line 13. Second, we simplified the condition of the loop in lines 10–13 of method `extractMin` (Fig. 3) to `temp != minNode`, which is equivalent because `findMinNode` yields a node with a minimal key. This change simplifies the specification of `findMinNode`. Third, we execute this loop only if `minNode` is not the first node of the list and otherwise set `this.Nodes` to `temp.sibling`. This change simplifies the loop invariant significantly. Besides these changes, the code we verified is a direct translation of the Java implementation. Note that we consider the second and third change to improve the clarity of the code, whereas the first change merely simplifies verification.

The manipulation of predicates via manual folding and unfolding gives a lot of control over the proof search, but introduces a very high overhead. Most of this overhead is needed to manipulate list segments during iterative traversals. Possible remedies are support for loops that are specified with pre- and postconditions instead of loop invariants [25], support for automatic folding and unfolding [3, 19], or a specification based on iterated separating conjunction rather than recursive predicates [15]. We plan to explore these options as future work.

Our solution requires 13 local assertions to make the SMT solver succeed. These local assertions provide expressions that trigger quantifier instantiations or temporarily unfold predicates to extract information from their bodies. Both usages should be automated.

Finally, the case study demonstrated the importance of a good development environment that parallelizes verification and caches verification results [13]. We have implemented both features since we completed the case study.

8 Conclusion

We presented the first solution to the Binomial Heap verification challenge from the VSComp 2014 competition. Our solution uses Viper's permission logic and makes heavy use of recursive predicates and heap-dependent abstraction functions.

The solution suggests several directions for future work. First, it requires a significant amount of annotations, especially loop invariants and ghost code to manipulate recursive predicates. It will be interesting to explore whether at least some of these annotations can be inferred. Second, our solution required 13 local assertions to help the SMT solver prove certain obligations; we will explore strategies to eliminate those. Third, we plan to encode the example using magic wands or iterated separating conjunction instead of list-segment predicates in order to reduce the overhead of manipulating these predicates.

This paper is a contribution to a book in honor of Arnd Poetzsch-Heffter's 60th birthday. One of my first scientific discussions with Arnd was about the role of object invariants in specifying object-oriented programs, especially, about the difference between invariants and well-formedness predicates as part of method specifications [20]. This discussion was the starting point of my PhD work and many more years of research on various forms of invariants. I hope Arnd will enjoy reading this case study about the verification of object invariants. I would like to take this opportunity to thank him for the tremendous support he has given me as my PhD advisor and beyond.

Acknowledgements We thank Marcelo Frias for proposing the Binomial Heaps verification challenge and for clarifying the results of previous verification attempts. We are grateful to Malte Schwerhoff for his help with the verification effort and to Alex Summers for the performance measurements.

References

1. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods (IFM)*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017.
2. J. Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
3. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages (POPL)*, pages 289–300. ACM, 2009.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.
5. M. Doko and V. Vafeiadis. Tackling real-life relaxed concurrency with FSL++. In H. Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*, pages 448–475. Springer, 2017.

6. C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In E. L. Miller and S. Hand, editors, *Symposium on Operating Systems Principles (SOSP)*, pages 1–17. ACM, 2015.
7. C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In J. Flinn and H. Levy, editors, *Operating Systems Design and Implementation (OSDI)*, pages 165–181. USENIX Association, 2014.
8. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In G. Castagna, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *LNCS*, pages 451–476. Springer, 2013.
9. S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Abstract read permissions: Fractional permissions without the fractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *LNCS*, pages 315–334, 2013.
10. J. Kaiser, H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In P. Müller, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 74 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
11. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiss. The 1st Verified Software Competition: Experience report. In M. Butler and W. Schulte, editors, *Formal Methods (FM)*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
12. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel. In J. N. Matthews and T. E. Anderson, editors, *Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.
13. K. R. M. Leino and V. Wüstholz. Fine-grained caching of verification results. In *Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 380–397. Springer, 2015.
14. P. Müller. The binomial heap verification challenge in Viper: Online appendix. viper.ethz.ch/onlineappendix-binomialheap, 2018.
15. P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV)*, volume 9779 of *LNCS*, pages 405–425. Springer, 2016.
16. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.
17. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic (CSL)*, pages 1–19. Springer, 2001.
18. M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *Principles of Programming Languages (POPL)*, pages 247–258. ACM, 2005.
19. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 711–728. Springer, 2014.
20. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Jan. 1997. https://softech.cs.uni-kl.de/homepage/en/publications.
21. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*. IEEE Computer Society Press, 2002.

22. M. Schwerhoff and A. J. Summers. Lightweight support for magic wands in an automatic verifier. In J. T. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 37 of *LIPIcs*, pages 614–638. Schloss Dagstuhl, 2015.
23. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
24. A. J. Summers and S. Drossopoulou. A formal semantics for isorecursive and equirecursive state abstractions. In G. Castagna, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *LNCS*, pages 129–153. Springer, 2013.
25. T. Tuerk. Local reasoning about while-loops. In R. Joshi, T. Margaria, P. Müller, D. Naumann, and H. Yang, editors, *VSTTE 2010. Workshop Proceedings*, pages 29–39. ETH Zurich, 2010.
26. V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 867–884. ACM, 2013.