Karlsruhe Institute of Technology

Bounded Verification of an Optimized Shortest Path Implementation

Studienarbeit

Michael Nagel

s_nagel2@atis.uka.de
michael.nagel@devzero.de

Department of Informatics
ITI - Institute of Theoretical Informatics

Reviewer:	Junior Prof. Dr. Mana Taghdiri
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	Dipl.-Inform. Julian Dibbelt

Duration: 01. January 2012 – 31. March 2012

Abstract

In this thesis we apply bounded program verification techniques to an implementation of Dijkstra's Algorithm optimized for execution speed.

Bounded program verification is a formal technique to build confidence in the correctness of a computer program. All possible inputs in a user-defined scope are guaranteed to be checked, but no statement is made about inputs outside the bounds. Dijkstra's Algorithm is a well-known algorithm to calculate shortest paths in graphs. One of its applications is in route planning to search the shortest route from Karlsruhe to Berlin in a graph representation of the European road network. The algorithm utilizes a priority queue often implemented using a binary heap.

We port Julian Delling's implementation of Dijkstra's Algorithm to Java because we use JForge and InspectJ for bounded program verification. This way we are able to verify important properties of the underlying BinaryHeap data structure. We report a bug in the implementation undiscovered by previous testing. Furthermore, we find two more bugs in our untested Java port of the codebase. We also discover many implicit assumptions on the implementation level.

The aforementioned results were obtained by a modular bottom-up approach. Originally, we used a top-down approach, characterizing valid inputs and criteria for correct outputs. We intended to verify the correctness of an implementation of Dijkstra's Algorithm within a bounded scope and increase the scope as far as possible without specifying the internals. Top-down verification of the `FlatBaseDijkstra.run()` method that calculates the shortest path between nodes in a graph did not succeed for several reasons: performance issues, complex analysis of counterexamples, and optimized code not created with verification in mind.

The analyzed implementation of Dijkstra's Algorithm was not specially created to be easy to verify; it uses complex data structures for the priority queue and graph representation and implements different optimization techniques important for real-world usage. For example, memory layout is taken into consideration in order to minimize cache footprint.

Contents

1. Introduction

On a precautionary note, be advised that the Java virtual machine heap used for dynamic memory allocation and the tree-based data structure called heap are two independent concepts and should not be confused.

We apply bounded program verification techniques to our Java port of an established implementation of Dijkstra's Algorithm [Dij59] that has been optimized for execution speed [Del09]. We verify important properties of the underlying BinaryHeap data structure using both JForge [Yes09] and InspectJ [LNT12]. We report a bug in the implementation undiscovered by previous testing. Furthermore, we find two more bugs in our untested Java port of the codebase.

All three bugs were detected using a modular bottom-up approach. The top-down approach, characterizing valid inputs and criteria for correct outputs on the top level, did not succeed because of performance issues and complex analysis of the counterexamples.

1.1. Bounded Program Verification

Bounded program verification is a formal technique to build confidence in the correctness of programs. It can be classified in between traditional testing and full formal verification when considering the trade-off between the achievable degree of confidence and the required effort for the user [Den09]. On the one hand, in bounded program verification all inputs up to a certain size are guaranteed to be considered. On the other hand, it executes fully automatically, requiring no interaction and only few annotations. Bounded program verification has a focus on data structure rich code, e.g. code written in object-oriented programming languages.

In bounded program verification, individual methods are analyzed. A method transforms a prestate (heap state and parameters) into a poststate (heap state and return value). The user defines preconditions that must hold in the prestate before executing the method, and postconditions that are expected to hold in the poststate after the method finishes execution. The process is modular in structure so verified properties of one method can be used to verify other methods.

Bounded program verification imposes further restrictions (bounds) on the considered prestates and execution length. Typically the following bounds are imposed: The bitwidth for integer numbers is limited, the number of instances (allocated objects) of each class is limited, and loops and recursive calls are unrolled a fixed number of times. Typical settings for the bounds might be 5-5-2 and bounded program verification makes no statement about inputs that cannot be processed within these bounds. On the one hand it is desirable to increase the bounds as far as possible. On the other hand increasing the bounds is expensive in terms of verification runtime. The small scope hypothesis [ADK03] tells us that good coverage is possible with low bounds.

For efficiency reasons, current bounded program verification tools encode the Java program and all specification into propositional logic formulas or satisfiability modulo theories problems. When preconditions are encoded as PRE, the program execution is encoded as $PROG$, postconditions are encoded as $POST$, and exceptions are signaled by $EXCEPT$ then the solvers search a satisfying assignment for $PRE \wedge PROG \wedge (\neg POST \vee EXCEPT)$. Thus, in bounded program verification, an unsatisfiable formula is a good thing because the formula is satisfied by counterexamples that violate the specification.

Bounded program verification tools typically execute in three phases. Preprocessing: code, specification, and bounds are transformed into a SAT or SMT formula. Solving: an off-the-shelf solver is used to solve the generated formula. Postprocessing: the output of the solvers is turned into a human-readable counterexample.

JForge [Yes09] is one of the most mature bounded program verification tools. JForge checks a subset of the Java language against a specification written in JFSL [Yes09], based on Alloy [Jac02]. The JForge backend relies on Alloy and a SAT solver, SAT4J [BP10] by default. In inline mode the code of all called submethods is inlined. This mode does not require specification of submethods but does not scale as well as constraint mode. In constraint mode the specification of the called submethod is considered instead of the actual code. Constraint mode requires additional specification effort from the user. Hybrid mode combines inline mode and constraint mode but the respective disadvantages are not solved completely.

InspectJ [LNT12] is an alternative to JForge and first versions became available during our work on this thesis. Specifications are written in a dialect of the Java Modeling Language (JML) [LBR99]. The InspectJ backend is based on SMT (Satisfiability Modulo Theories) and the Z3 solver [dMB08]. Initial benchmarks [LNT12] show that InspectJ performs better than JForge, especially for big scopes.

The verification workflow is similar with both tools. It is an iterative process: we start with a functional requirement of the analyzed program and annotate the relevant code with the formalized specification. Then we run the bounded program verification tool and analyze any counterexample it might produce. Fixing the problem causing the counterexample can require changes to the code and/or specification. After every change the bounded program verification tool has to be rerun to check if the problem is fixed and/or if there are regressions.

1.2. Dijkstra's Algorithm

Dijkstra's Algorithm is one of the best known algorithms in connection with graphs and computes single-source shortest paths in a graph with non-negative edge weights. It was originally published by Dijkstra in 1959 [Dij59].

In Dijkstra's Algorithm all nodes are initialized with a tentative distance of infinity from the source node. The graph is then explored starting at the source node, moving further and further away from it. Nodes are examined (settled) with monotonically increasing distance from the source node. When a node is settled a new node might be discovered (touched for the first time) or a shorter path to an unsettled node might be discovered. In either case the tentative distance to the respective node is updated. The algorithm ends when all reachable nodes have been settled.

The performance of an implementation of Dijkstra's Algorithm is highly depending on the underlying priority queue data structure. An efficient way to track the tentative distances is crucial for a fast implementation. In sparse graphs like road networks the BinaryHeap data structure offers optimal performance of $O(|E| + |V| \cdot \log |V|)$.

The BinaryHeap is a commonly used tree-based data structure that stores a set of key/-value pairs and satisfies the following property:

$key(p) \leq key(n)$
for each node n (except the root node) and its parent node p.

For a priority queue the key of the elements in the BinaryHeap is used as priority (or distance in the case of Dijkstra's Algorithm, with a node in the graph as value). The BinaryHeap allows efficient extraction of the value with the minimal key. The most important operations to be executed on a BinaryHeap are:

- `insert()`: insert an element into the BinaryHeap
- `deleteMin()`: remove the element with the smallest key from the BinaryHeap
- `decreaseKey()`: decrease the key of an element in the BinaryHeap

There are two important internal functionalities to restore the heap property after an element changed:

- `upheap()` to move a small element upwards to its correct position
- `downheap()` to move a large element downwards to its correct position

It is intuitive to think of the BinaryHeap as a tree, but many implementations (including the one discussed in this thesis) store a BinaryHeap as an array for performance reasons with implicit storage of parents and children.

1.3. Our Approach

It was an explicit goal of this project to verify an existing implementation optimized for real-world usage. The code is based on an implementation of Dijkstra's Algorithm originally written by Daniel Delling in C++ for his Ph.D. thesis on route planning in road networks [Del09]. It uses complex data structures for the priority queue and graph representation and implements different optimization techniques resulting in a speedup in relevant route planning scenarios.

JForge [Yes09] is the most mature bounded program verification tool available but it works with Java code. An effort to create a version of JForge that operates on C and possibly C++ code is stalled. Consequently, the C++ code had to be ported to Java in order to work with JForge. Large parts of the codebase could be re-used verbatim because both C++ and Java are curly braces programming languages.

The implementation does not provide any specification, which had to be created from scratch. Functional level specification is the most fundamental and in many ways the most important specification. Ideally verification would succeed if the user specified that the solution returned by Dijkstra's Algorithm is the shortest possible path between the corresponding nodes. Formalizing this requirement already makes it more verbose, but unfortunately that is insufficient because JForge and InspectJ still return counterexamples.

Functional level specification is typically what is mentioned when talking about what has been verified in a program. However, additional specification is required. Bounded program verification tools are excellent at detecting usage of pointers that are not checked against null, array indexes that are out of bounds, arithmetic overflow and aliasing issues. Implementation level specification takes care of all these corner cases that make programs crash under certain circumstances but are irrelevant in the big picture. Context level specification is about making the verification process work with the tools at hand. Bounded program verification tools systematically consider every bit pattern within a given scope as a possible prestate. However, many random bit patterns do not represent valid program states.

We tried two different approaches to create the specification. Top-Down verification handles the algorithm as a black box. Only the input and output of the top level method, `FlatBaseDijkstra.run()`, is considered. It is irrelevant how the output is calculated. Relatively few specifications are added and they are on the functional level. Bellman's Principle of Optimality can be used to specify that the result of the `FlatBaseDijkstra.run()` method is indeed a shortest path. The somewhat lengthy JML applying the principle to Dijkstra's Algorithm can be found in chapter 4.3.1.1. There is also one functional level precondition in Dijkstra's Algorithm: all edge weights must be non-negative.

Bottom-Up verification means verifying the submethods called by the top-level method separately. This allows the usage of constraint mode but requires far more extensive context specification. It also helps to build confidence in the methods and provides simpler counterexamples if there are problems.

The most important invariant for the BinaryHeap is the min heap property (see chapter 2.2.1)—even though it is not strictly on the functional level, as the external functionality (being a priority queue) could be implemented otherwise.

1.4. Thesis Organization

The remainder of this thesis is structured as follows: Chapter 2 introduces the relevant background concepts for the unfamiliar reader, chapter 3 lists prior work related to Dijkstra's Algorithm and verification in general as well as the verification of Dijkstra's Algorithm in particular, chapter 4 discusses the approach executed in this thesis, chapter 5 discusses the various results and findings we gathered, and chapter 6 concludes, with some outlook in chapter 7.

2. Background

This chapter provides a brief introduction to the concepts used later on; it explains the basics of bounded program verification, and discusses two current tools, namely JForge and InspectJ. We review Dijkstra's Algorithm and the heap data structure used as a priority queue in many implementations of Dijkstra's Algorithm.

2.1. Bounded Program Verification

Bounded program verification is a formal technique to build confidence in the correctness of programs. It has a focus on data structure rich code, e.g. code written in object-oriented programming languages. Bounded program verification can be classified in between traditional testing and full formal verification when considering the trade-off between the achievable degree of confidence and the required effort for the user [Den09]. On the one hand, in bounded program verification all inputs up to a certain size are guaranteed to be considered. On the other hand, it executes fully automatically, requiring no interaction and only few annotations.

The following is intended as a quick introduction with concrete examples for unfamiliar readers. More precise theoretical background information can be found in Gregory D. Dennis' Ph.D. thesis [Den09].

In bounded program verification individual methods are analyzed. The process is modular in structure so verified properties of one method can be used to verify other methods. A method transforms a prestate (heap state and parameters) into a poststate (heap state and return value). The user defines any number of preconditions that must hold in the prestate before executing the method, and postconditions that are expected to hold in the poststate after the method finishes execution. Usually it is also possible to define invariants for complete classes that basically serve as a pre- and postcondition for methods in that class.

Calling a method when the preconditions for that method are met should result in a program state where the postconditions for that method hold. If this is the case for every prestate fulfilling the preconditions, the program is verified to conform to its specification. Bounded program verification imposes further restrictions (bounds) on the considered prestates and execution length. Programs are said to be verified with respect to those bounds. Typically the following bounds are imposed:

1. The bitwidth for integer numbers is limited.

2. The number of instances (allocated objects) of each class is limited.
3. Loops and recursive calls are unrolled a fixed number of times.

The semantics of such a program are nearly identical to the semantics of the original program restricted to runs that can execute within the specified bounds. Hence problems with the bounded program are useful indicators of problems with the real program.

On the one hand it is desirable to increase the bounds as far as possible. Some bugs are missed when the bounds are too low and whole branches can become unreachable. On the other hand increasing the bounds is expensive in terms of verification runtime. The small scope hypothesis [ADK03] tells us that good coverage is possible with low bounds. In any case, it is important to balance the bounds. Creating a lot of instances might be without effect if loops are not unrolled to handle all instances.

Typical settings for the bounds might be 5-5-2, i.e. integers are five bits wide and overflow occurs at 16, classes are limited to five instances (with further instantiations causing the program to go out of bounds), and all loops are unrolled twice (ensuring the loop would not be entered a third time, going out of bounds otherwise). Bounded program verification makes no statement about inputs that cannot be processed within these bounds.

Current bounded program verification tools encode the Java program and all specification into propositional logic formulas or satisfiability modulo theories problems. Existing off-the-shelf SAT (or SMT) solvers solve the formulas faster than executing all considered inputs sequentially.

If integer numbers are limited to a bitwidth of 3, every integer Java variable, e.g. `x, y` and `z`, can be encoded using three boolean variables, one for each bit. A boolean Java variable, e.g. `b`, can be encoded using one boolean variable in the resulting formula. Field access, array access, object instantiation and all other Java features are encoded using similar requirements for the underlying boolean values. Control flow is simplified by unrolling all loops, inlining all method calls, and converting the code to single static assignment form.

For example, the Java statement

```
int x = b ? y : z;
```

results in the propositional formula

$$(b^i \wedge (x_1^{i+1} = y_1^i) \wedge (x_2^{i+1} = y_2^i) \wedge (x_3^{i+1} = y_3^i)) \vee (\neg b^i \wedge (x_1^{i+1} = z_1^i) \wedge (x_2^{i+1} = z_2^i) \wedge (x_3^{i+1} = z_3^i))$$

where the superscript i is used to number variable versions incrementally.

We can assume that the preconditions including the invariants are encoded as PRE, the program execution is encoded as $PROG$ and the postconditions including the invariants are encoded as $POST$. Exceptions are signaled by $EXCEPT$. The solvers then search a satisfying assignment for

$$PRE \wedge PROG \wedge (\neg POST \vee EXCEPT)$$

If such an assignment is found (i.e. the formula is satisfiable) this means that the programs definitely does not conform to its specification and the assignment is returned as a counterexample.

In bounded program verification, an unsatisfiable formula is a good thing. If the formula is unsatisfiable it means the program conforms to its specification. However, the formula might be unsatisfiable for undesirable reasons: It might be that PRE or $PRE \wedge PROG$ alone are already unsatisfiable. In the first case the contradiction in the precondition needs to be fixed, whereas in the latter case the bounds need to be increased because the program cannot be executed within the bounds at all, for example because the program creates more objects than the bounds allow.

It is a good idea to check if $PRE \land PROG \land POST \land \neg EXCEPT$ is satisfiable to ensure there is at least one valid execution within the specified bounds. $PRE \land PROG$ must be satisfiable (see above) and the remaining reasons for unsatisfiability are the following. $POST$ might be unsatisfiable meaning that a contradiction in the postcondition needs to be fixed. Alternatively, it might be that there is a mismatch between code and specification, and all prestates in the current scope violate the postcondition or lead to exceptional behavior. The bounded program verification tool cannot tell if the code or the specification (or both) is broken, so the user has to detect and resolve the conflict.

Bounded program verification tools typically execute in three phases:

1. **Preprocessing**: In the preprocessing phase the inputs (i.e. program, specification, and bounds) are transformed into a SAT or SMT formula.
2. **Solving**: An off-the-shelf solver such as SAT4J [BP10] or Z3 [dMB08] is used to solve the formula generated in the preprocessing phase.
3. **Postprocessing**: In the postprocessing phase the tools transform the output of the solvers into a human-readable counterexample. This might include the input leading to the problematic execution as well as a trace with information on what exactly went wrong where exactly.

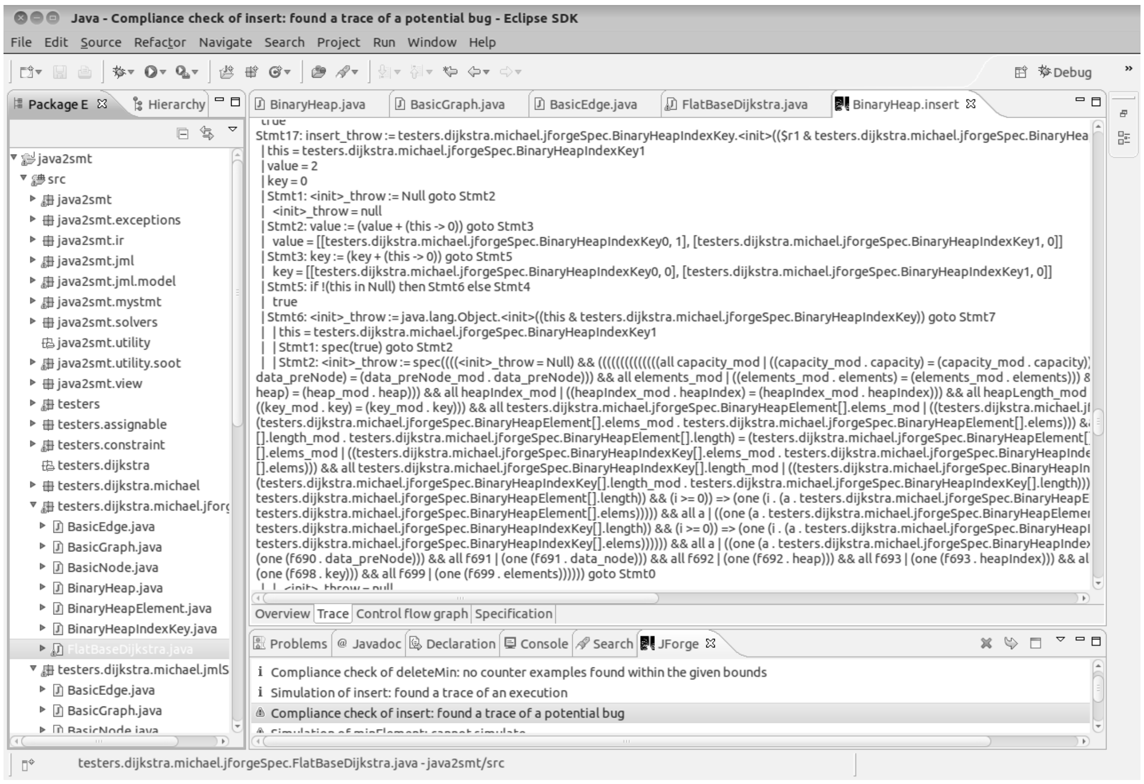
2.1.1. JForge

JForge [Yes09] is one of the more mature bounded program verification tools. JForge checks a subset of the Java language against a specification written in JFSL [Yes09], a custom specification language based on Alloy [Jac02]. The JForge frontend is an Eclipse plug-in allowing the user to set options, trigger the verification and examine the postprocessed output. The JForge backend relies on Alloy and a SAT solver, SAT4J [BP10] by default.

JForge offers three operating modes:

- **Inline Mode**: In inline mode the code of all called submethods is inlined. Advantages: Inline mode does not require specification of submethods. This allows to check partial specifications describing just some properties of the code. Disadvantages: Inline mode does not scale well.
- **Constraint Mode**: In constraint mode the precondition of the called submethod is checked when a submethod with specification is called. If it holds, the program state is changed non-deterministically to make it respect the specified postcondition of the called submethod. The actual code in the called submethod is ignored. Advantages: Constraint mode scales better than inline mode. Disadvantages: Very detailed postconditions are required for replaced submethods. Otherwise traces inconsistent with the actual code are generated. See chapter 5.2.3 for a more detailed discussion of this problem.
- **Hybrid Mode**: Hybrid mode combines inline mode and constraint mode. Some advantages from both modes apply, but as none of the respective disadvantages are solved for all cases we found it to be not too useful in practice.

A screenshot of JForge can be seen in figure 2.1.



```
Stmt17: insert_throw := testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey.<init>(($r1 & testers.dijkstra.michael.jforgeSpec.BinaryHea
 | this = testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey1
 | value = 2
 | key = 0
 | Stmt1: <init>_throw := Null goto Stmt2
 |   <init>_throw = null
 | Stmt2: value := (value + (this -> 0)) goto Stmt3
 |   value = [[testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey0, 1], [testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey1, 0]]
 | Stmt3: key := (key + (this -> 0)) goto Stmt5
 |   key = [[testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey0, 0], [testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey1, 0]]
 | Stmt5: if !(this in Null) then Stmt6 else Stmt4
 |   true
 | Stmt6: <init>_throw := java.lang.Object.<init>((this & testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey)) goto Stmt7
 |   | this = testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey1
 |   | Stmt1: spec(true) goto Stmt2
 |   | Stmt2: <init>_throw := spec((((<init>_throw = Null) && ((((((((((((((all capacity_mod | ((capacity_mod . capacity) = (capacity_mod . capacity))
data_preNode) = (data_preNode_mod . data_preNode))) && all elements_mod | ((elements_mod . elements) = (elements_mod . elements))) &
heap) = (heap_mod . heap))) && all heapIndex_mod | ((heapIndex_mod . heapIndex) = (heapIndex_mod . heapIndex))) && all heapLength_mod
((key_mod . key) = (key_mod . key))) && all testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[].elems_mod | ((testers.dijkstra.michael.j
(testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[].elems_mod . testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[].elems))) &
[].length_mod . testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[].length) = (testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[
[].elems_mod | ((testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey[].elems_mod . testers.dijkstra.michael.jforgeSpec.BinaryHeapInde
[].elems))) && all testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey[].length_mod | ((testers.dijkstra.michael.jforgeSpec.BinaryHeapIn
(testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey[].length_mod . testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey[].length)))
testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[].length)) && (i >= 0)) => (one (i . (a . testers.dijkstra.michael.jforgeSpec.BinaryHeapE
testers.dijkstra.michael.jforgeSpec.BinaryHeapElement[].elems))))) && all a | ((one (a . testers.dijkstra.michael.jforgeSpec.BinaryHeapElemen
testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey[].length)) && (i >= 0)) => (one (i . (a . testers.dijkstra.michael.jforgeSpec.BinaryHeapI
testers.dijkstra.michael.jforgeSpec.BinaryHeapIndexKey[].elems)))))) && all a | ((one (a . testers.dijkstra.michael.jforgeSpec.BinaryHeapIndex
(one (f690 . data_preNode))) && all f691 | (one (f691 . data_node))) && all f692 | (one (f692 . heap))) && all f693 | (one (f693 . heapIndex))) && al
(one (f698 . key))) && all f699 | (one (f699 . elements)))))) goto Stmt0
```

```
i Compliance check of deleteMin: no counter examples found within the given bounds
i Simulation of insert: found a trace of an execution
⚠ Compliance check of insert: found a trace of a potential bug
```

```
testers.dijkstra.michael.jforgeSpec.FlatBaseDijkstra.java - java2smt/src
```

Figure 2.1.: Screenshot of JForge showing the trace for a counterexample.

2.1.2. InspectJ

First versions of InspectJ [LNT12] became available during our work on this thesis. InspectJ is a command line tool with features comparable to JForge. InspectJ checks a subset of the Java language against specifications written in a dialect of the Java Modeling Language (JML) [LBR99]. In contrast to JForge, the InspectJ backend is based on SMT (Satisfiability Modulo Theories) and the Z3 solver [dMB08].

Initial benchmarks show that InspectJ performs better than JForge, especially for big scopes. InspectJ still lacks some features that JForge supports, including the postprocessing of the output, but the tool is under active development and rapidly improving.

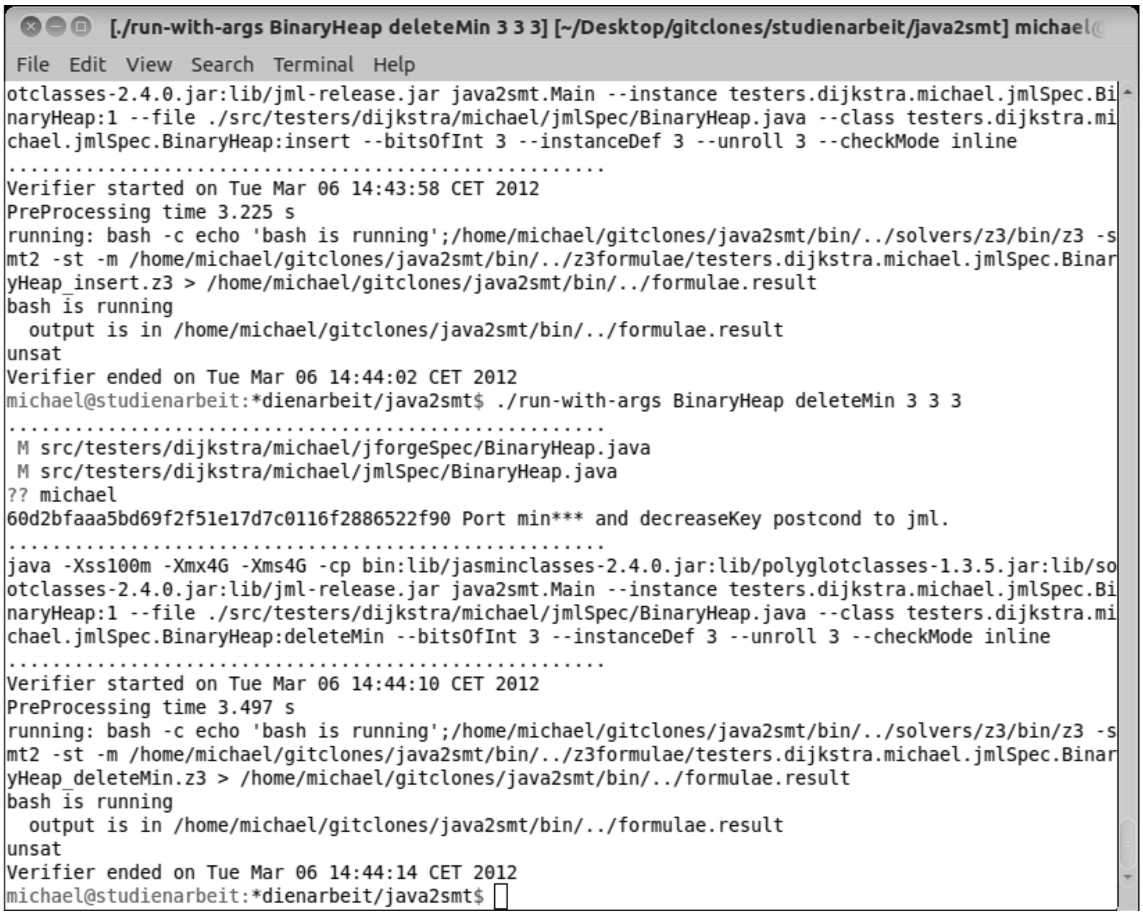
InspectJ operates in inline mode.

A screenshot of InspectJ can be seen in figure 2.2.

2.2. Dijkstra's Algorithm

Dijkstra's Algorithm is one of the best known algorithms in connection with graphs and computes single-source shortest paths in a graph with non-negative edge weights. It was originally published by Dijkstra in 1959 [Dij59].

A pseudocode implementation of Dijkstra's Algorithm is listed below.


[./run-with-args BinaryHeap deleteMin 3 3 3] [~/Desktop/gitclones/studienarbeit/java2smt] michael
File Edit View Search Terminal Help

```
otclasses-2.4.0.jar:lib/jml-release.jar java2smt.Main --instance testers.dijkstra.michael.jmlSpec.Bi
naryHeap:1 --file ./src/testers/dijkstra/michael/jmlSpec/BinaryHeap.java --class testers.dijkstra.mi
chael.jmlSpec.BinaryHeap:insert --bitsOfInt 3 --instanceDef 3 --unroll 3 --checkMode inline
....................................................
Verifier started on Tue Mar 06 14:43:58 CET 2012
PreProcessing time 3.225 s
running: bash -c echo 'bash is running';/home/michael/gitclones/java2smt/bin/../solvers/z3/bin/z3 -s
mt2 -st -m /home/michael/gitclones/java2smt/bin/../z3formulae/testers.dijkstra.michael.jmlSpec.Binar
yHeap_insert.z3 > /home/michael/gitclones/java2smt/bin/../formulae.result
bash is running
  output is in /home/michael/gitclones/java2smt/bin/../formulae.result
unsat
Verifier ended on Tue Mar 06 14:44:02 CET 2012
michael@studienarbeit:*dienarbeit/java2smt$ ./run-with-args BinaryHeap deleteMin 3 3 3
....................................................
 M src/testers/dijkstra/michael/jforgeSpec/BinaryHeap.java
 M src/testers/dijkstra/michael/jmlSpec/BinaryHeap.java
?? michael
60d2bfaaa5bd69f2f51e17d7c0116f2886522f90 Port min*** and decreaseKey postcond to jml.
....................................................
java -Xss100m -Xmx4G -Xms4G -cp bin:lib/jasminclasses-2.4.0.jar:lib/polyglotclasses-1.3.5.jar:lib/so
otclasses-2.4.0.jar:lib/jml-release.jar java2smt.Main --instance testers.dijkstra.michael.jmlSpec.Bi
naryHeap:1 --file ./src/testers/dijkstra/michael/jmlSpec/BinaryHeap.java --class testers.dijkstra.mi
chael.jmlSpec.BinaryHeap:deleteMin --bitsOfInt 3 --instanceDef 3 --unroll 3 --checkMode inline
....................................................
Verifier started on Tue Mar 06 14:44:10 CET 2012
PreProcessing time 3.497 s
running: bash -c echo 'bash is running';/home/michael/gitclones/java2smt/bin/../solvers/z3/bin/z3 -s
mt2 -st -m /home/michael/gitclones/java2smt/bin/../z3formulae/testers.dijkstra.michael.jmlSpec.Binar
yHeap_deleteMin.z3 > /home/michael/gitclones/java2smt/bin/../formulae.result
bash is running
  output is in /home/michael/gitclones/java2smt/bin/../formulae.result
unsat
Verifier ended on Tue Mar 06 14:44:14 CET 2012
michael@studienarbeit:*dienarbeit/java2smt$
```

Figure 2.2.: Screenshot of InspectJ.

```
 1 input: a graph of nodes and edges with non-negative length
 2
 3 all nodes are initialized
 4   - with a tentative distance of infinity
 5   - untouched (there is no known path yet)
 6   - unsettled (final distance is not yet known)
 7
 8 set tentative distance of the source node to 0
 9 mark the source node as touched
10
11 repeat until there are no more touched-but-unsettled nodes:
12   u <- touched-but-unsettled node of minimal tentative distance
13   d <- tentative distance of u
14   mark u as settled
15   repeat for each neighbor v of u:
16     len <- length of edge from u to v
17     if (d+len) < (tentative distance of v):
18       set tentative distance of v to d+len
19       mark v as touched
20
21 output: length of shortest path for each reachable node
```

In lines 3-6, all nodes are assigned a tentative distance of infinity from the source node. After this initialization Dijkstra's Algorithm explores the graph starting at the source node, moving further and further away from it. In lines 11-14 nodes are examined (settled) with monotonically increasing distance from the source node. When a node is settled a new node might be discovered (touched for the first time) in line 17 or a shorter path to an unsettled node might be discovered. In either case the tentative distance to the respective node is adjusted in lines 16-19. Once a node is settled its tentative distance becomes final. Any path constructed later on passes through a node that is even further away and thus the path cannot be shorter. The path corresponding to the calculated length can be restored later if in line 19 **v** remembers **u** as its predecessor.

In the simplest case Dijkstra's Algorithm calculates the distance from one source node to all reachable nodes. This mode is called one-to-all. If only the distance to one particular target node is relevant (one-to-one) the outer loop can be terminated when the target node is about to be settled because the tentative distance becomes final then. Note that the search is not specifically directed towards the target: in euclidean space the search space is explored in expanding spheres around the source node.

An efficient way to track the touched nodes and their tentative distances is crucial for a fast implementation of Dijkstra's Algorithm. The worst case performance of an implementation of Dijkstra's Algorithm is highly depending on the underlying priority queue data structure used to store, query and update that information. In a graph $G(V, E)$ with nodes V and edges E the worst case performance ranges from $O(|E| + |V|^2)$ using a linear list to $O(|E| + |V| \cdot \log |V|)$ using a Fibonacci heap and a reverse mapping from nodes to heap position. In sparse graphs like road networks the simpler BinaryHeap data structure also offers optimal performance of $O(|E| + |V| \cdot \log |V|)$.

2.2.1. BinaryHeap

A priority queue stores a set of key/value pairs. It allows efficient extraction of the value with the minimal key. Priority queues can be implemented using a binary minimum heap—a commonly used tree-based data structure that satisfies the following property:

Definition 1 *minimum heap property:*
$key(p) \leq key(n)$
for each node n (except the root node) and its parent node p.

In this thesis BinaryHeap refers to a binary minimum heap because that is what is used in the analyzed implementation. The most important operations to be executed on a BinaryHeap are:

- `insert()`: insert an element into the BinaryHeap
- `deleteMin()`: remove the element with the smallest key from the BinaryHeap
- `decreaseKey()`: decrease the key of an element in the BinaryHeap

It is intuitive to think of the BinaryHeap as a tree, but many implementations (including the one discussed in this thesis) store a BinaryHeap as an array for performance reasons. If the root element is stored at index 1, the children of element **e** at index i are stored at indexes $2 * i$ and $2 * i + 1$, if they exists. The parent of element **e** is stored at index $\lfloor i/2 \rfloor$ except for the root node. The parent/child relationship is implicit and not stored explicitly. See figure 2.3 for an illustration.

Decreasing the key of an element **e** or inserting a new element **e** at the end of the array may require to move element **e** upwards in the BinaryHeap. This restructuring method is called `upheap()` and is necessary because the heap property might be violated after decreasing

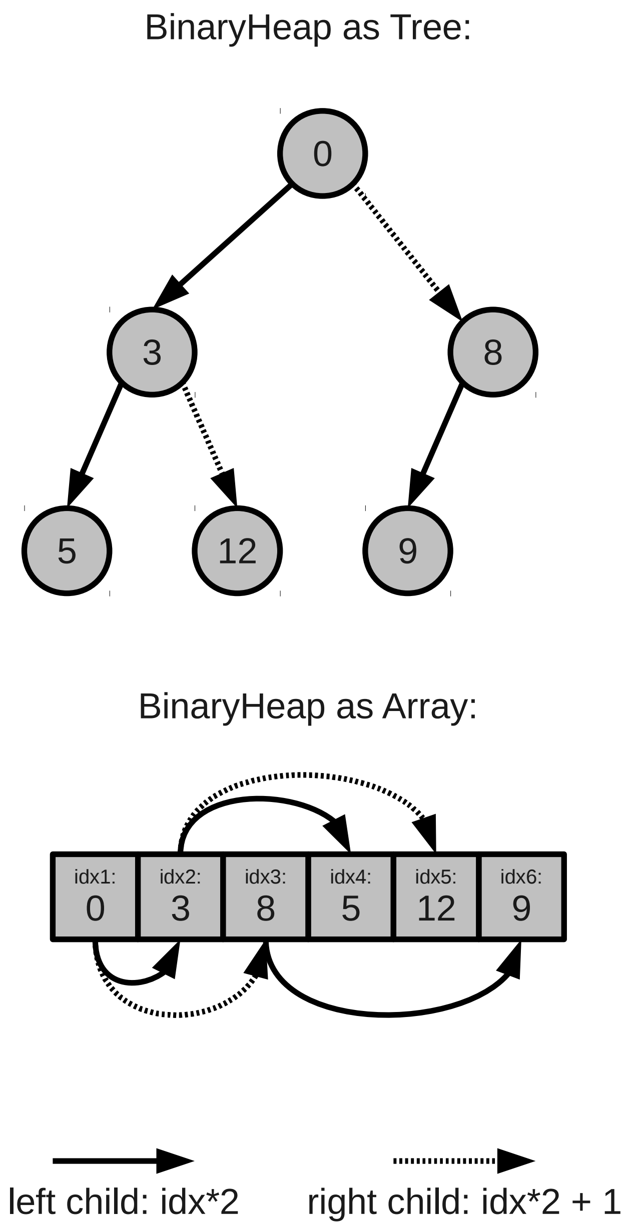


Figure 2.3.: Different representations of a BinaryHeap.
The numbers in the elements are keys; values are not pictured.

the key or inserting **e** at the end. `upheap()` is implemented by repeatedly exchanging **e** with its respective parent until the parent has a smaller key than **e** or the root node is reached.

After removing the minimal element (i.e. the root element) from a BinaryHeap at index 1, the last element **e** from the BinaryHeap is moved to index 1 to fill the gap. This typically results in a temporary violation of the heap property that is corrected by `downheap()`. During `downheap()` **e** is repeatedly exchanged with its respective child with the smaller key, until **e**'s key is smaller than the keys of both of **e**'s children or the bottom of the heap is reached.

Elements store a key and a value. The value may be a reference to elsewhere with the real value. For a priority queue the key of the elements is used as priority (or distance in the case of Dijkstra's Algorithm, with a node in the graph as value).

3. Related Work

3.1. Software Verification

Software testing already is an established technique in the software industry. Software verification is a promising alternative and/or supplement and a lot of different approaches to software verification exist.

In this thesis we concentrated on bounded program verification because it can handle data structure rich programs and check them exhaustively (in a bounded scope) without requiring too much user interaction.

JForge

JForge [Yes09] is one of the more mature bounded program verification tools. JForge checks a subset of the Java language against a specification written in JFSL [Yes09], a custom specification language based on Alloy [Jac02]. The JForge frontend is an Eclipse plug-in allowing the user to set options, trigger the verification and examine the postprocessed output. The JForge backend relies on Alloy and a SAT solver, SAT4J [BP10] by default.

JForge is used extensively in this thesis.

InspectJ

During our work on this thesis first versions of InspectJ [LNT12] became available. InspectJ is a command line tool with features comparable to JForge. InspectJ checks a subset of the Java language against specifications written in a dialect of the Java Modeling Language (JML) [LBR99]. In contrast to JForge, the InspectJ backend is based on SMT (Satisfiability Modulo Theories) and the Z3 solver [dMB08].

Initial benchmarks show that InspectJ performs better than JForge, especially for big scopes. InspectJ still lacks some features that JForge supports, including the postprocessing of the output, but the tool is under active development and rapidly improving.

JForge is used extensively in this thesis.

3.2. Dijkstra's Algorithm

Dijkstra's Algorithm is one of the best known algorithms in connection with graphs and it is the topic of many publications and countless implementations exists.

A Note on Two Problems in Connexion with Graphs

The original paper originally published by Dijkstra in 1959 [Dij59] describing the algorithm.

Introduction to Algorithms

Dijkstra's Algorithm is discussed in the reputable Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein [CLRS01].

Boost

Dijkstra's Algorithm is part of the Boost C++ libraries [boo12].

Engineering and Augmenting Route Planning Algorithms

The code verified in this thesis is based on an implementation of Dijkstra's Algorithm originally written by Daniel Delling in C++ for his Ph.D. thesis on route planning in road networks [Del09]. The code was developed with raw performance in mind, and no plans to verify it. It uses complex data structures and applies many optimizations resulting in speedup in relevant route planning scenarios.

3.3. Verifying Dijkstra's Algorithm

Several attempts have been made to verify Dijkstra's Algorithm. The following section lists previous attempts and describes how they are different from the approach described in this thesis.

KeY: Dijkstra's Algorithm

In his diploma thesis Volker Klasen proves the correctness of a Java implementation of Dijkstra's Algorithm in KeY [Kla10, ABB$^+$05]. However, the implementation of the algorithm is specifically created to make it provable. For example, a linear list is traversed instead of using a BinaryHeap, which is slower in theory and practice. KeY was not able to automatically close the required proof obligations, so besides creating detailed specification a significant amount of time was spend on manually constructing a proof.

Somewhat related, Peter H. Schmitt constructed a proof of the correctness of the Bellman Equation [Sch11] used in the specification.

Jahob: Dijkstra's Algorithm

Robin Mange and Jonathan Kuhn verified some unmentioned properties of a basic implementation of Dijkstra's Algorithm using Jahob [MK07, jah12]. The code is minimal, academic code without optimizations. In comparison to our approach Jahob requires more detailed annotations, e.g. loop invariants. Also, it is unclear if Jahob could handle more complex code, as the authors mention performance issues with the code they verified.

Mizar: Dijkstra's Algorithm

Jing-Chao Chen formalized Dijkstra's Algorithm in Mizar and proved it [Che03, miz12]. His work is very theoretical and does not involve an actual implementation.

ACL2: Dijkstra's Algorithm

J Strother Moore and Qiang Zhang verified an unoptimized Lisp implementation of Dijkstra's Algorithm using ACL2 [MZ05, KMB97]. The proof is not automatic and requires significant user interaction.

4. Our Approach

Three things are required to perform a bounded program verification of an implementation of Dijkstra's Algorithm: the actual implementation, a specification of the implementation, and a bounded program verification tool to perform the verification.

There are many existing implementations of Dijkstra's Algorithm available, thus starting a new implementation is unnecessary—on the contrary, it was an explicit goal of this project to verify an existing implementation optimized for real-world usage. The implementation used does not provide any specification, which had to be created from scratch. JForge and an early version of InspectJ were used as bounded program verification tools.

This chapter describes the origins of the analyzed implementation of Dijkstra's Algorithm, and how we ported it to Java for verification purposes. We tried two different approaches creating the specification: a top-down approach without looking at the internals, and a bottom-up approach, starting with the verification of single submethods. Finally we describe the iterative workflow of testing and improving specification.

4.1. Implementation of Dijkstra's Algorithm

The code verified in this thesis is based on an implementation of Dijkstra's Algorithm originally written by Daniel Delling in C++ for his Ph.D. thesis on route planning in road networks [Del09]. The implementation was benchmarked for multiple theses and publications on algorithm engineering by Delling and other researchers, e.g. Sascha Meinert [Mei11]. The code was developed with raw performance in mind, and no plans to verify it. It uses complex data structures and applies many optimizations resulting in speedup in relevant route planning scenarios.

After stripping all functionality except the shortest path calculation from the codebase, the following classes remain:

- `FlatBaseDijkstra` (contains the main algorithm)
- `BinaryHeap` (the heap implementation)
- `BinaryHeapElement` (data container)
- `BinaryHeapIndexKey` (data container)
- `BasicGraph` (data container)

- `BasicNode` (data container)
- `BasicEdge` (data container)

The names are taken from the Java port for consistency throughout this thesis.

Typical usage of the `FlatBaseDijkstra` class is structured like this:

1. An instance of `BasicGraph` is created. Typically the graph is read from a file, but Input/Output is ignored in this thesis.
2. The `FlatBaseDijkstra` constructor is called and the `BasicGraph` instance created in step 1 is passed as a parameter.
3. `FlatBaseDijkstra.run()` is called and the source and target nodes are passed as parameters. Dijkstra's Algorithm is executed and makes numerous calls to different methods of an internal `BinaryHeap` instance.
4. `FlatBaseDijkstra.getDistance()`, an accessor method for the distance calculated and stored within the `BinaryHeap` instance, is called.

In the BinaryHeap class an array of `IndexKey`s builds a heap, where each node stores its key and a reference to a `HeapElement`. The `HeapElement`s store the real payload and are kept in a separate array, that does not conform to the heap property; in fact `HeapElement`s are never relocated once added. This way less data needs to be moved when the heap is restructured. `IndexKey`s are kept small to minimize cache footprint: for example indexes into the `HeapElement` array are stored instead of pointers, because unlike pointers indexes can be stored as 32bit integers on 64bit machines; more of them will fit in the cache and/or less data needs to be discarded from the cache. The key is stored in the `HeapElement`, so it can be accessed after removal of the `IndexKey` from the heap. Nonetheless a copy of the key is kept in the `IndexKey` to save a dereference operation and to minimize cache footprint when traversing the heap. No virtual methods are used, only templating.

A dummy `IndexKey` is kept as a guard object in the heap at all times. The parent/child relationship is never expressed explicitly, only indexes are multiplied/divided by 2 when necessary.

Proper usage of all classes is assumed; parameters are rarely checked, overflow of path lengths is not handled and the code easily crashes if invoked outside the context of `Flat-BaseDijkstra.run()`.

4.1.1. Java Port of the Codebase

There was an effort to create a version of JForge that operates on C and possibly C++ code at the Toshiba Corporate Research and Development Center [DY12]. No code, documentation or paper is available. We contacted the engineers and the project seems dormant.

Consequently the C++ code had to be ported to Java in order to work with JForge. Large parts of the codebase could be re-used verbatim because both C++ and Java are curly braces programming languages.

There have been some more involving changes, however:

- Code not directly related to the calculation of shortest paths was removed. This affects debugging code to visually trace single steps of the algorithm, code to read graphs and queries from files, and similar code. For example the following code was completely dropped:

```
//visualization (can be set by compilerflag -DVISUAL)
#ifdef VISUAL
#define VISUALMODE(x) x
#endif
#ifndef VISUAL
#define VISUALMODE(x)
#endif
```

These changes are justified because we are interested in verifying the core of the implementation—not the supporting code.

- The original code uses typedefs and templating to allow reuse in other search algorithms. In our port all numerical types become `int`, and instead of function or class templates, we create versions with the actually used types substituted for the template parameters. For example we make the following replacement:

```
template < typename ExternalKey,
typename MetaExtKey,
typename Data,
typename Count,
typename Key = ExternalKey,
typename KeyExtractor = SimpleKeyExtractor<ExternalKey> >
class BinaryHeap { ...
```

becomes

```
class BinaryHeap { ...
```

Typedefs and templates are not supported in Java and the flexibility is not required for the analyzed implementation.

- `std::vector<...>` is replaced with plain arrays as can be seen in the following example:

```
typedef BinaryHeapElement<ExternalKey,Data,Count> PQElement;
std::vector<PQElement> _elements;
```

becomes

```
public BinaryHeapElement[] elements;
```

`std::vector<...>` from the C++ Standard Template Library is not available in Java. The implementations of the `java.util` interface `List<...>` cause problems with the bounded program verification tools, because they require support for generics and the Java class library. This change causes no further problems because the number of nodes in the graph is a reasonable upper limit for the number of elements in the heap.

We introduced one bug when implementing this change, see chapter 5.2.1.

The port does not include a `main` method, so it can not be run and it was—on purpose—not tested in a traditional way, to see what errors the bounded program verification tools detect. Some syntax errors where detected and fixed by the IDE, Eclipse. There are no changes on the algorithmic level between the C++ and the Java implementation.

4.2. Three Levels of Specification

Ideally verification would succeed if the user specified that the solution returned by Dijkstra's Algorithm is the shortest possible path between the corresponding nodes. Formalizing this requirement already makes it more verbose (see chapter 4.3.1.1), but unfortunately that is insufficient because the bounded program verification tools still return counterexamples.

Additional specification is required to get rid of the counterexamples and to successfully verify the program. From a user's point of view this additional specification is different from the original specification. Bounded program verification tools are oblivious and do not make any distinction.

For the user the following levels of specification exist:

1. **Functional Level Specification:** This specification is the most fundamental and in many ways the most important specification. It describes the desired effect of a method, and applies at a high level. It could be used to specify pseudocode or structures described by UML diagrams. Functional level specification is typically what is mentioned when talking about what has been verified in a program.
2. **Implementation Level Specification:** This specification works around small shortcomings in the actual implementation and makes it airtight. The specification or ideally the code must take care of all corner cases that make programs crash under certain circumstances but are irrelevant in the big picture. Bounded program verification tools are excellent at detecting usage of pointers that are not checked against null, array indexes that are out of bounds, arithmetic overflow and aliasing issues.
3. **Context Level Specification:** This level of specification is about making the verification process work with the tools at hand. Bounded program verification tools systematically consider every bit pattern within a given scope as a possible prestate. However, many random bit patterns do not represent valid program states—states that can be reached starting with an empty heap and running code from the program. Most programs function according to *garbage in, garbage out* and will not deal with such cases properly. As verification tools cannot tell what prestates should be considered valid, the developer must explicitly state it in specification. The counterexamples excluded by context level specification look similar to counterexamples ruled out by implementation level specification. However, they are only an issue in the verification process, and never during program execution.

   Furthermore, there is specification to cope with shortcomings of the tools: For example in the current version of InspectJ it is necessary to add an additional precondition `array != null && array[5] != null` when there already is a precondition `array[5].key == 5` that should imply the former.

Sometime it helps to temporarily introduce specification to influence the counterexample that is removed again later on. JForge and InspectJ only report one counterexample at a time—and not necessarily a simple one at that. The scope of the counterexample is bounded but the bounds are traversed in no particular order. So a complex counterexample with many interconnected instances in the prestate might be created. One might ask if this can also happen when the prestate is simple before untangling that prestate. The only influence on the counterexamples one has is to add further preconditions that exclude complex prestates.

The following self-contained scenario shows the different levels of specification and typical bugs found by formal verification:

In a racing simulation the `race` class stores an array `a` of `car` instances to represent the competing drivers and their cars. The array is ordered according to the current position in the race. The method `overtake()` is called whenever the car at position `pos` overtakes another car. `overtake()` restores the correct order of `a` and increases the counter of maneuvers for the involved cars.

The following method does all this and would actually work perfectly in the context of the racing simulation without ever being the source of any problems with the program:

```
1 void overtake(int pos) {
2   car[] a = this.a;
3   car tmp = a[pos];
4   a[pos]   = a[pos+1];
5   a[pos+1] = tmp;
6   a[pos].count   += 1;
7   a[pos+1].count += 1;
8 }
```

A formal specification of the `overtake()` method at the functional level looks this:

```
1 /*@ ensures(
2   @ a[pos]         == \old(a[pos+1]) &&
3   @ a[pos+1]       == \old(a[pos])   &&
4   @ // tricky: array rearrangement and counting is intertwined
5   @ a[pos].count   == \old(a[pos+1].count) + 1 &&
6   @ a[pos+1].count == \old(a[pos].count)   + 1
7   @ ); @*/
```

Line 2 of the specification ensures that the car that was at position `pos+1` in the prestate (denoted by `old`) is at position `pos` in the poststate. Line 3 ensures that the other car was moved respectively. Lines 5 and 6 ensure that for both cars `count` is increased by one.

However—and this might come as a surprise—the code does not conform to this specification because of the following assumptions that most likely hold in the given scenario but are not true in general, especially if the method is verified in isolation.

1. In line 3 `a` is assumed to be not `null`. However, `a` might be `null`, leading to a `null` dereference, resulting in a `NullPointerException`. Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.

2. In line 3 `pos` is assumed to be zero or positive. However, `pos` might be negative, leading to an array access that is out of bounds, resulting in an `ArrayIndexOutOfBoundsException`. Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.

3. In line 3 `pos` is assumed to be smaller than `a.length`. However, `pos` might be `a.length` or bigger, leading to an array access that is out of bounds, resulting in an `ArrayIndexOutOfBoundsException`. Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.

4. In line 4 `pos` is assumed to be smaller than `Integer.MAX_VALUE`. However, `pos` might be `MAX_VALUE`, leading to an integer overflow when calculating `pos+1`, resulting in undesired behavior later. Depending on the rest of the program this either requires

implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.

5. In line 4 `pos+1` is assumed to be smaller than `a.length`. However, `pos+1` might be `a.length` or bigger, leading to an array access that is out of bounds, resulting in an `ArrayIndexOutOfBoundsException`. Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.
6. In line 6 `a[pos]` (initially `a[pos+1]`) is assumed to be not `null`. However, `a[pos]` might be `null`, leading to a `null` dereference, resulting in a `NullPointerException`. Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.
7. In line 6 `a[pos].count` is assumed to be smaller than `Integer.MAX_VALUE`. However, `a[pos].count` might be `MAX_VALUE`, leading to an integer overflow when adding `1`, resulting in undesired behavior later. This requires implementation level specification describing what should happen instead of overflow.
8. In line 7 `a[pos+1]` is assumed to be not `null`. However, `a[pos+1]` might be `null`, leading to a `null` dereference, resulting in a `NullPointerException`. Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.
9. In line 7 `a[pos+1].count` is assumed to be smaller than `Integer.MAX_VALUE`. However, `a[pos+1].count` might be `MAX_VALUE`, leading to an integer overflow when adding `1`, resulting in undesired behavior later. This requires implementation level specification describing what should happen instead of overflow.
10. `a[pos]` and `a[pos+1]` are assumed to be referencing different instances of `car`. However, `a[pos]` and `a[pos+1]` might be aliased increasing `count` twice in lines 5 and 7, possibly resulting in unexpected behavior later.

    Aliasing means that one entity is referred to by two different names. This is not necessarily a problem, but causes problems when a compiler or a programmer assumes that two *different* entities are referred. Aliasing can happen when working with multiple pointers or references, especially with different array elements if they are by-reference as in Java, with the same array element if two differently named indexes (with the same value) are used, with overlapping arrays when referred to by pointers in C, and with overlapping memory segments when using `memmove()` et al.

    Depending on the rest of the program this either requires implementation level specification enforcing a valid value or context level specification excluding unreachable prestates.
11. As a consequence of the aliasing issue above it follows, that `a[pos].count` is assumed to be smaller than `Integer.MAX_VALUE-1`, because otherwise adding `1` twice will overflow, possibly resulting in unexpected behavior later.
12. The specification is incomplete in the sense that an implementation that incorrectly sets all elements of `a` to null, except those at index `pos` and `pos+1` would still be verified. This can be worked around using modifies clauses, but constitutes a major problem in constraint mode, see chapter 5.2.3.
13. Let us assume one more invariant: the sum of the counts of all cars in one race is an even number (because it is always increased by `2`). After all problems mentioned above have been fixed, and the aliasing problem is solved by requiring all cars in a

race to be different, there will still be counterexamples with a second instance `i` of the `race` class, where one car is added to both races. Even though the invariant is correct in the context of `this` it might still fail in the context of `i` if only one of the cars has been added to that race.

The code cannot be verified successfully unless it handles all those corner cases gracefully—the snippet above does not—or unless another set of specifications is added, making all the implicit assumptions explicit.

Implementation level specification undoubtedly refers to real deficits in the codebase. It is possible to write actual snippets of code (e.g. test cases) that trigger the problems. An attacker might exploit the problem or the problem might show when the code is used under slightly changed circumstances. System programmers and developers of security relevant code typically are interested in these corner cases and handle them in their code.

However, a lot of other code (including the implementation of Dijkstra's Algorithm discussed in this thesis) is not written so defensively because it is used only by non-malicious programmers knowing the implicit assumptions.

Missing implementation and context level specification both completely block further verification. The tools only report one counterexample at a time, even if it is of low value to the programmer. As long as there are low level counterexamples one cannot tell whether or not there still are problems on the functional level. Therefore it is not possible to postpone low level specification to a later point.

For successful bounded program verification each and every arithmetic operation must be guaranteed to not overflow, each and every reference dereferenced must be guaranteed to be not `null` and each and every array access must be guaranteed to be within bounds. Each and every reference must be assumed to be aliased.

4.3. Top-Down Approach

JForge and InspectJ will only check for some built-in properties, namely the absence of `NullPointerException` and `ArrayIndexOutOfBoundsException`, if no specification is provided. In order to check functional properties of the code user-defined specification is required.

Top-Down verification handles the algorithm as a black box. Only the input and output of the top level method, `FlatBaseDijkstra.run()`, is considered. It is irrelevant how the output is calculated. Relatively few specifications are added and they are on the functional level.

4.3.1. Postconditions of Dijkstra's Algorithm

The goal of this thesis is to prove that *the result of the* `FlatBaseDijkstra.run()` method is indeed a shortest path. To check this with the bounded program verification tools we need a criterion that is true if and only if a given path is a shortest path.

4.3.1.1. Bellman's Principle of Optimality

In dynamic programming a more complex problem is solved by combining solutions of less complex problems. Richard Bellman formulated the following principle in the context of dynamic programming:

Definition 2 *Bellman's Principle of Optimality:*
An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

The solution to the complex problem is as good as the solutions of the simpler problems combined. In terms of the shortest path problem, where we want to prove that Dijkstra's Algorithm is an optimal policy, every path from **s** to **t** with $\mathbf{s} \neq \mathbf{t}$ can be split in two parts:

1. part: path from **s** to **u**
2. part: path from **u** to **t**

where **u** is a direct neighbor of **t**. The shortest path from **s** to **t** via **u** cannot be shorter than those two parts combined (added). This holds for all neighbors **u** of **t**. **u** will be an alias for **s** if **t** is a neighbor of **s**.

Formalized:

$\forall t \in V : \forall u \in N(t) : D(s,t) \leq D(s,u) + d(u,t)$

- V is the set of nodes in the graph.
- $N(a)$ is the set of the neighbors (nodes that are connected via a direct edge) of a.
- $D(a,b)$ is the distance (length of the shortest path) from a to b
  and assumed to be ∞ if b is not reachable from a.
- $d(a,b)$ is the length of the edge from a to b
  and only defined if there is a direct edge from a to b.

If $\mathbf{s} \neq \mathbf{t}$ there must be at least one node where equality holds, namely any node **u** that is a direct predecessor to **t** on a shortest path from **s** to **t**:

$\forall t \in V \setminus \{s\} : \exists u \in N(t) : D(s,t) = D(s,u) + d(u,t)$

The source node has distance 0 by definition: $D(s,s) := 0$.

In JML syntax the formula for the one-to-all case reads:

```
/*@ ensures (
  @ (this.queue.elements
            [this.graph.nodes[sourceNodeId].slotId].key == 0) &&
  @ (\forall int targetId;
  @  ((0 <= targetId) &&
  @   (targetId < this.graph.nodes.length-1) &&
  @   (targetId != sourceNodeId))
  @  ==>
  @  (\exists int neighborId;
  @   ((0 <= neighborId) &&
  @    (neighborId < this.graph.nodes.length-1))
  @   &&
  @   (\exists int edgeId;
  @    (this.graph.nodes[neighborId].firstEdge <= edgeId) &&
  @    (edgeId < this.graph.nodes[neighborId+1].firstEdge) &&
  @    (this.graph.edges[edgeId].targetNodeId == targetId) &&
  @    (this.queue.elements
                [this.graph.nodes[targetId].slotId].key
  @     ==
  @     this.queue.elements
                [this.graph.nodes[neighborId].slotId].key
  @     + this.graph.edges[edgeId].weight)
  @   )
  @   &&
  @   (\forall int edgeId;
  @    (this.graph.nodes[neighborId].firstEdge <= edgeId) &&
  @    (edgeId < this.graph.nodes[neighborId+1].firstEdge) &&
  @    (this.graph.edges[edgeId].targetNodeId == targetId) ==>
  @    (this.queue.elements
                [this.graph.nodes[targetId].slotId].key
  @     <=
  @     this.queue.elements
                [this.graph.nodes[neighborId].slotId].key
  @     + this.graph.edges[edgeId].weight)
  @   )
  @  )
  @ )); @*/
```

There are still some implementation level issues that have to be addressed:

- Overflow: Preconditions make sure there is no overflow, see chapter 4.3.2.
- Multiple Connected Components: The specification is correct for graphs with multiple connected components if the distance is set to ∞ for unreachable nodes. In the implementation `Integer.MAX_VALUE` can be used if instead of `+` an addition method that does not overflow is used. A node will have distance ∞ if and only if all neighbor nodes have distance ∞ (are unreachable), thus it is semantically is correct. `this.queue.elements[this.graph.nodes[nodeId].slotId].key` might be undefined for untouched nodes, thus a wrapper that returns ∞ is required.
- Search Mode: The specification only works in one-to-all mode because all shortest distances need to be known. For one-to-one mode it must be checked that distances

are correct for settled nodes, tentative distances are correct considering the settled nodes and all nodes that should have been settled actually have been settled.

- Edge Direction: The specification assumes that all edges are bidirectional. The code supports directed graphs via `forward/backward` flags in the edges. To support this in the specification, it must check whether the forward flag is set and skip the edge otherwise.

Please note that we did not successfully verify this specification with JForge or InspectJ. It might therefore be incomplete or inaccurate.

4.3.2. Preconditions of Dijkstra's Algorithm

There is one functional level precondition in Dijkstra's Algorithm: all edge weights must be non-negative. In JML this reads:

```
/*@ requires(
  @ (\forall int i; ((i >= 0) && (i < this.graph.edges.length))
  @ ==> this.graph.edges[i].weight >= 0)
  @ ); @*/
```

There are some implementation level preconditions describing how a valid instance of the graph looks.

```
/*@ requires(
  @ (this.graph.nodes != null) &&
  @ (this.graph.nodes.length >= 2) && // dummy
  @ (\forall int i; ((i >= 0) && (i < this.graph.nodes.length))
  @ ==>
  @ ((this.graph.nodes[i] != null) &&
  @  (this.graph.nodes[i].firstEdge >= 0) &&
  @  ((i != this.graph.nodes.length-1) ==>
      (this.graph.nodes[i].firstEdge
       <
       this.graph.edges.length)) &&
  @  (this.graph.nodes[i].timestamp <= this.time)))
  @ ); @*/
```

```
/*@ requires(
  @ (this.graph.edges != null) &&
  @ (this.graph.edges.length >= 0) &&
  @ (\forall int i; ((i >= 0) && (i < this.graph.edges.length))
  @ ==>
  @ ((this.graph.edges[i] != null) &&
  @  (this.graph.edges[i].targetNodeId >= 0) &&
  @  (this.graph.edges[i].targetNodeId <
        this.graph.nodes.length) &&
  @  (this.graph.edges[i].forwardFlag == true))) // WORKAROUND
  @ ); @*/
```

`node.firstEdge` must be in order for the loop traversing all edges:

```
/*@ requires(
  @ (this.graph.nodes[0].firstEdge == 0) &&
  @ (\forall int i; ((i >= 0) && (i < this.graph.nodes.length-1))
  @ ==>
  @ ((this.graph.nodes[i].firstEdge <=
        this.graph.nodes[i+1].firstEdge)))
  @ &&
  @ (this.graph.nodes[this.graph.nodes.length -1].firstEdge ==
        this.graph.edges.length) // dummy points behind
  @ ); @*/
```

Obvious bounds checking:

```
/*@ requires(
  @ (0 <= sourceNodeId) &&
  @ (sourceNodeId < this.graph.nodes.length-1) && // dummy
  @ (0 <= targetNodeId) &&
  @ (targetNodeId < this.graph.nodes.length-1) && // dummy
  @ (this.queue.capacity >= this.graph.nodes.length-1)
  @ ); @*/
```

The verifiers are excellent at finding arithmetic overflows. The algorithm basically sums up the lengths of a set of edges that form a path. This addition is prone to overflow. The original code does not even consider that case because it is intended to be run on road networks where lengths of shortest paths are small in comparison to `Integer.MAX_VALUE`.

This vague domain knowledge must be formalized in an exact precondition to be useful to the bounded program verification tools. The precondition must hold in a degenerated road network where e.g. all nodes are linked linearly.

We require that the sum of the length of all edges does not overflow. As no edge is added to a path more than once, the summation can no longer overflow.

Unfortunately InspectJ does not support aggregation of values via a sum predicate. As a workaround we create a `maxWeight` variable that limits the weight of any single edge.

```
/*@ requires(
  @ (\forall int i; ((i >= 0) && (i < this.graph.edges.length))
  @ ==>
  @ ((i * maxWeight <= Integer.MAX_VALUE) &&
  @  (i * maxWeight > 0)) // catch overflow
  @ (\forall int i; ((i >= 0) && (i < this.graph.edges.length))
  @ ==>
  @ (this.graph.edges.weight <= maxWeight)
  @ ); @*/
```

For performance reasons (fast reverse lookup), the position in the BinaryHeap is stored within each element. An element cannot be added to two BinaryHeap instances **h1** and **h2**, as only the position in one BinaryHeap can be stored in the element. However there are possible prestates where one element is in multiple BinaryHeap instances at the exact same position, and running the algorithm will corrupt the other BinaryHeap instances. A counterexample will be reported if there are invariants that catch such corruptions. This never happens in the context of Dijkstra's Algorithm. As a workaround only one instance of the BinaryHeap class is allowed.

In JFSL this can be achieved by:

```
one BinaryHeap
```

In JML there is no language level support for this feature and an additional command line parameter has to be given to InspectJ.

4.4. Bottom-Up Approach

Bottom-Up verification means verifying the submethods called by `FlatBaseDijkstra.run()` separately. This allows the usage of constraint mode. It also helps to build confidence in the methods and provides simpler counterexamples if there are problems.

4.4.1. Postconditions for BinaryHeap

The most important invariant for the BinaryHeap is the min heap property (see chapter 2.2.1)—even though it is not strictly functional level, as the external functionality (being a priority queue) could be implemented otherwise.

In this context and in JML the min heap property reads:

```
// MIN-HEAP PROPERTY
/*@ invariant (
 @ (\forall int i;
 @  ((i/2 >= 0) &&
 @   (i >= 0) &&
 @   (i < this.heapLength) &&
 @   (i < this.heap.length))
 @  ==>
 @  ((this.heap[i] != null) &&
 @   (this.heap[i/2] != null) &&
 @   (this.heap[i].key >= this.heap[i/2].key)))
 @ ); @*/
```

The public methods of BinaryHeap and their functional level postconditions are:

- BinaryHeap constructor: After running the BinaryHeap constructor, the BinaryHeap must be empty. This is the case if `heapLength == 1` because of the dummy element that is always present.

```
//@ ensures (this.heapLength - 1 == 0);
```

- `BinaryHeap.insert()` method: After running `insert()` a new element must have been added with the correct key.

```
/*@ ensures (
  @ (this.elementsLength ==
  @    \old(this.elementsLength + 1)) &&
  @ (this.elements[\result].key == key)
  @ ); @*/
```

- `BinaryHeap.decreaseKey()` method: After running `decreaseKey()` the key for the element must have been decreased.

```
//@ ensures(this.elements[slotId].key == newKey);
```

- `BinaryHeap.min()` method: The `min()` method must return the minimal key in the heap.

```
/*@ ensures(
  @ (\forall int i; ((i >= 1) && (i < this.heapLength))
  @ ==> \result <= this.heap[i].key) &&
  @ (\exists int j; ((j >= 1) && (j < this.heapLength))
  @ ==> \result == this.heap[j].key)
  @ ); @*/
```

Together with the min heap property this can be simplified to:

```
//@ ensures (\result == this.heap[1].value);
```

- `BinaryHeap.minElement()` method: The `minElement()` method must return the minimal element in the heap, identified by the index in the `elements` array.

```
/*@ ensures (
  @ (\result == this.heap[1].value) &&
  @ (\exists int res; (\result == this.heap[res].value) &&
  @ (\forall int j; ((j >= 1) && (j < this.heapLength))
  @ ==> (this.heap[res].key <= this.heap[j].key)))
  @ ); @*/
```

We use the temporary variable `res` because `\result` seems to be invalid in some syntactical constellations.

- `BinaryHeap.deleteMin()` method: The `deleteMin()` method must remove the minimal element from the heap.

```
/*@ ensures(
  @ (\exists int res;
  @ ((res == \result) &&
  @  (this.elements[res].heapIndex == 0) &&
  @  (\old(this.elements[res].heapIndex != 0)) &&
  @  (\forall int j; ((j >= 1) && (j < this.heapLength))
  @   ==> (this.elements[res].key <= this.heap[j].key))))
  @ ); @*/
```

- `BinaryHeap.clear()` method: After running the `clear()` method, the BinaryHeap must be empty. A `heapLength` of 1 means that the BinaryHeap is empty because of the dummy element that is always present.

```
//@ ensures (this.heapLength - 1 == 0);
```

- `BinaryHeap.size()` method: The `size()` method must return the number of elements in the heap, corrected for the dummy element.

```
//@ ensures (\result == this.heapLength - 1);
```

- `BinaryHeap.empty()` method: The `empty()` method must return true if and only if the heap is empty, i.e. there are no elements besides the dummy element.

```
//@ ensures (\result == (this.heapLength - 1 == 0));
```

4.5. Verification Workflow

Bounded program verification is an iterative process:

1. Start with a functional requirement of the analyzed program.

2. Annotate the according method with the formalized specification.
3. Run the bounded program verification tool. The runtime is highly depending on the selected bounds and can range from less than one second to hours or days.
4. Analyze the counterexample. First, trace the control flow if all statements are executed as expected. Second, untangle the prestate to get it back in terms of Java objects. Sometimes it is enough to translate the obviously relevant state, but unexpected aliasing might make it necessary to translate all state. This step can well take hours and greatly benefits from tool support.
5. Change specification or code to resolve the problem leading to the counterexample. Repeat steps 3 and 4 until no more counterexample show up.
6. Check if the specification works as intended. Deliberately insert a bug it should catch. There might be a contradiction in the preconditions, the annotation might get parsed in an unexpected way, the tools might be buggy or not support a certain feature, or the selected scope might be too low.

5. Evaluation

This chapter discusses the bugs and implicit assumptions discovered while executing the approach described in chapter 4. The findings from the top-down approach are discussed first, followed by the findings from the bottom-up approach. Benchmarks were executed to assess performance of the tools. We conclude with some remarks regarding the feature set and usability of the tools and there is a list of suggested improvements.

5.1. Results of the Top-Down Approach

The top-down approach handles `FlatBaseDijkstra` as a black box, trying to directly verify `FlatBaseDijkstra.run()`. After initial experiments it became clear that this would not succeed for two reasons:

1. performance: in inline mode JForge cannot handle `FlatBaseDijkstra.run()` within a reasonable scope. See chapter 5.4 for reasonable scopes, and see chapter 5.3 for actual performance. Using constraint mode might improve performance but requires specification of the called methods; i.e. the BinaryHeap.
2. counterexamples: with small scopes counterexamples are found that do not directly violate the functional specification. These counterexample are about crashes in corner cases hidden in lots of irrelevant state. To understand these counterexamples, an understanding of the implicit assumptions in the code (chapter 5.2.2) is required, obtained by analyzing the BinaryHeap.

`FlatBaseDijkstra.run()` repeatedly calls public methods of BinaryHeap. A correct implementation of BinaryHeap is a prerequisite for Dijkstra's Algorithm, even if this approach violates the original top-down approach.

5.2. Results of the Bottom-Up Approach

The bottom-up approach requires a lot of additional specification that is unnecessary in top-down mode because the `FlatBaseDijkstra.run()` arranges everything accordingly. In top-down verification many preconditions for the BinaryHeap methods are required before they can be checked in isolation. With this additional specification we discovered bugs and recovered lots of undocumented knowledge about the BinaryHeap data structure.

5.2.1. Discovered Bugs

JForge and later InspectJ created counterexamples pointing out the following bug in the Java port that was also present in the C++ version. The bug has been reported to the original developers and is fixed.

```
// code with bug 'invalid memory access'
int minSlotId = minElement();
int lastHeapIndex = this.heapLength - 1;
BinaryHeapIndexKey lastIK = heap[lastHeapIndex];

this.dropHeap(); // remove last element
heap[1] = lastIK; // copy last element to front

int pivotHeapIndex = 1;
BinaryHeapIndexKey pivotIK = heap[pivotHeapIndex];
int droppingSlotId = pivotIK.value;
int droppingKey = pivotIK.key;
```

Under certain circumstances (when there is exactly one element in the heap) the element that is about to be moved to the front is removed by `dropHeap()`. There is no code handling this special case. As a result freed memory is accessed. The solution is to postpone freeing until after the last access:

```
// code without bug 'invalid memory access'
int minSlotId = minElement();
int lastHeapIndex = this.heapLength - 1;
BinaryHeapIndexKey lastIK = heap[lastHeapIndex];

heap[1] = lastIK; // copy last element to front

int pivotHeapIndex = 1;
BinaryHeapIndexKey pivotIK = heap[pivotHeapIndex];
int droppingSlotId = pivotIK.value;
int droppingKey = pivotIK.key;

this.dropHeap(); // remove last element
```

This bug was not discovered by testing, because in the C++ version it is unlikely that the memory is reallocated in the meantime.

The following two bugs were also discovered by JForge and InspectJ:

```
// code with bug 'aliasing issue 1'
BinaryHeapIndexKey bestChild = heap[nextIndex];
...
// child+parent are not exchanged. cyclic rotation instead.
heap[pivotHeapIndex] = bestChild; // pull up the entry


...

pivotIK = heap[pivotHeapIndex];
pivotIK.value = droppingSlotId; // insert dropping element here
pivotIK.key = droppingKey;
```

```
// code with bug 'aliasing issue 2'
BinaryHeapIndexKey ik = heap[heapIndex];
int key = ik.key;
int risingSlotId = ik.value;

... heapIndex changes ...

ik = heap[heapIndex];
ik.value = risingSlotId;
ik.key = key;
elements[risingSlotId].heapIndex = heapIndex;
```

The `IndexKey` in question is aliased in the Java version and must not be changed.

There is a difference between assignment of structs in `std::vector<...>` in C++ on the one hand and assigning Objects in a Java array on the other hand. The former is by value whereas the latter is by reference:

```
// C++
struct pair {
  int a;
  int b;
}
std::vector<pair> vec;
... // insert 2 pairs into vec

pair p = new pair(12, 34);
vec[0] = p;
vec[1] = p;

vec[0].a = 42;

// vec[1].a == 12; // unchanged
```

is different from

```
// JAVA
class pair{
  int a;
  int b;
}
pair[] vec = new pair[100];
... // insert 2 pairs into vec

pair p = new pair(12, 34);
vec[0] = p;
vec[1] = p;

vec[0].a = 42;

// vec[1].a == 42; // it changed!
```

In both cases the bug was introduced when porting from C++ to Java. The fixed versions are listed below.

```
// code without bug 'aliasing issue 1'
BinaryHeapIndexKey bestChild = heap[nextIndex];
...
// child+parent are not exchanged. cyclic rotation instead.
heap[pivotHeapIndex] = bestChild; // pull up the entry

...

// reuse some unused IK
// (and magically it even already contains the correct values!)
heap[pivotHeapIndex] = lastIK;
```

```
// code without bug 'aliasing issue 2'
BinaryHeapIndexKey ik = heap[heapIndex];
int key = ik.key;
int risingSlotId = ik.value;

... heapIndex changes ...

heap[heapIndex] = ik;
// adjust element to have valid back-pointer
elements[risingSlotId].heapIndex = heapIndex;
```

5.2.2. Recovered Implicit Assumptions

The bugs listed in section 5.2.1 break functionality even if the code is used as intended. Besides that we discovered a lot of cases where some implicit and undocumented assumption is made. From a developers point of view it might be argued that the code works as intended in the given context. From a verification perspective these are bugs where the external interface is unsafe, and might become a problem in the future. Reuse of the code in another context or slight changes in the calling context might possibly trigger the hidden bug.

Verification is only successful if there are no counterexamples at all. Therefore all of the listed issues are blockers to successful verification on the functional level.

- Invariants on the class level

  The following invariants enforce that the arrays `elements` and `heap` are not `null`, which is assumed basically everywhere and causes crashes if it is not the case. This is always the case in the context of `FlatBaseDijkstra.run()`, making this specification only necessary in bottom-up mode when checking methods out of context.

  ```
  // not null
  //@ invariant (this.elements != null);
  //@ invariant (this.heap != null);
  ```

  Arrays are used to simulate the functionality of `std::vector<..>` because the bounded program verification tools do not support Java Generics. The following invariants are necessary to rule out unreachable prestates. Entries in `heap` and `elements` are used starting at the front and later entries must be `null`. This is always the case in the context of `FlatBaseDijkstra.run()`, making this specification only necessary in bottom-up mode when checking methods out of context.

```
// capacity of the simulated vector
//@ invariant (this.capacity < Integer.MAX_VALUE);
//@ invariant (this.capacity > 0);

// array length
//@ invariant (this.elements.length == this.capacity + 1);
//@ invariant (this.heap.length == this.elements.length);

// tracker for heap length
//@ invariant (this.heapLength >= 1);
//@ invariant (this.heapLength <= this.capacity + 1);

// tracker for elements length
//@ invariant (this.elementsLength >= 1);
//@ invariant (this.elementsLength <= this.capacity + 1);
```

```
// unused array entries must be null
/*@ invariant (
  @ (\forall int i;
  @  ((i >= 0) &&
  @   (i < this.elements.length) &&
  @   (i >= this.elementsLength))
  @   ==> (this.elements[i] == null))
  @ ); @*/

/*@ invariant (
  @ (\forall int i;
  @  ((i >= 0) &&
  @   (i < this.heap.length) &&
  @   (i >= this.heapLength))
  @   ==> (this.heap[i] == null))
  @ ); @*/
```

This invariant defines that the dummy element must have key 0, placing it in front of all other elements. Otherwise problems arise with elements with keys smaller than the dummy key, but 0 is a guaranteed minimum. `FlatBaseDijkstra.run()` does not add elements with negative keys and uses a dummy element with `key` 0, making this specification only necessary in bottom-up mode when checking methods out of context.

```
// key for dummy element
//@ invariant (this.heap[0].key == 0);
```

The following invariants block counterexamples with `null` entries in `this.elements` or `this.heap`. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
/*@ invariant (
  @ (\forall int i;
  @   ((i >= 0) &&
  @    (i < this.elements.length) &&
  @    (i < this.elementsLength))
  @    ==> (this.elements[i] != null))
  @ ); @*/
```

```
/*@ invariant (
  @ (\forall int i;
  @  ((i >= 0) &&
  @   (i < this.heap.length) &&
  @   (i < this.heapLength))
  @   ==> (this.heap[i] != null))
  @ ); @*/
```

`heapIndex` must be an actually valid index. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
// elements must have valid heapIndex'es
/*@ invariant (
  @ (\forall int i;
  @  ((i >= 0) &&
  @   (i < this.elementsLength) &&
  @   (i < this.elements.length))
  @  ==>
  @  ((this.elements[i] != null) &&
  @   (this.elements[i].heapIndex >= 0)
  @   (this.elements[i].heapIndex < this.heapLength)))
  @ ); @*/
```

`FlatBaseDijkstra.run()` never adds an element to the heap multiple times. In a bottom-up approach this must be formalized. A prestate with one element added multiple times causes breakage when the heap is restructured. Only the properly linked element is updated, leaving behind a broken element. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
// elements cannot be added multiple times
// because they only store one pointer into the heap
/*@ invariant (
  @ (\forall int i, j;
  @  ((i >= 0) &&
  @   (j > i) &&
  @   (j < this.elementsLength) &&
  @   (j < this.elements.length))
  @   ==> (this.elements[i] != this.elements[j]))
  @ ); @*/
```

```
// same for the heap
/*@ invariant (
  @ (\forall int i, j;
  @  ((i >= 0) &&
  @   (j > i) &&
  @   (j < this.heapLength) &&
  @   (j < this.heap.length) &&
  @   (i < this.heap.length))
  @   ==> (this.heap[i] != this.heap[j]))
  @ ); @*/
```

Entries from `heap` and `elements` must be interconnected properly, linking forth and back again. See figures 5.1 and 5.2 for an illustration. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
// STRUCTURAL HEAP SANITY
/*@ invariant (
  @ (\forall int i;
  @  ((i >= 0) &&
  @   (i < this.heapLength) &&
  @   (i < this.heap.length))
  @   ==> ((this.heap[i] != null) &&
  @   (this.heap[i].value >= 0) &&
  @   (this.heap[i].value < this.elements.length) &&
  @   (this.elements[this.heap[i].value] != null) &&
  @   (this.elements[this.heap[i].value].heapIndex == i)))
  @ ); @*/
```

Linked `BinaryHeapElements` and `IndexKeys` must have matching keys. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
// KEYS MATCH
/*@ invariant (
  @ (\forall int i;
  @  ((i >= 0) &&
  @   (i < this.heap.length) &&
  @   (i < this.heapLength))
  @   ==> ((this.heap[i] != null) &&
  @   (this.heap[i].value >= 0) &&
  @   (this.heap[i].value < this.elements.length) &&
  @   (this.elements[this.heap[i].value] != null) &&
  @   (this.heap[i] != null) &&
  @   (this.elements[this.heap[i].value].key ==
  @      this.heap[i].key)))
  @ ); @*/
```

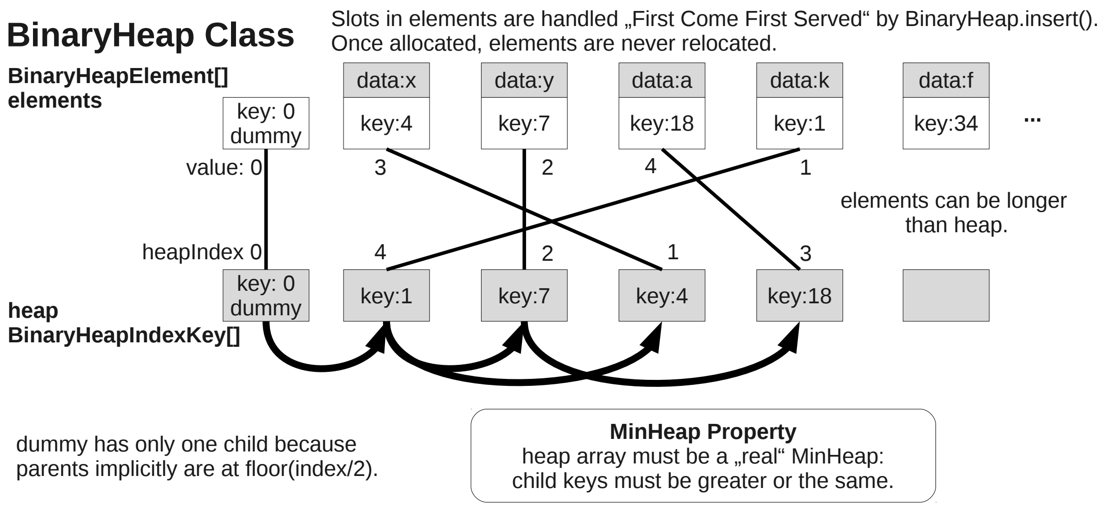

BinaryHeap Class
Slots in elements are handled „First Come First Served“ by BinaryHeap.insert(). Once allocated, elements are never relocated.
BinaryHeapElement[] elements
key: 0 dummy
data:x key:4
data:y key:7
data:a key:18
data:k key:1
data:f key:34
...
value: 0
3
2
4
1
elements can be longer than heap.
heapIndex 0
4
2
1
3
key: 0 dummy
key:1
key:7
key:4
key:18
heap BinaryHeapIndexKey[]
dummy has only one child because parents implicitly are at floor(index/2).
MinHeap Property
heap array must be a „real“ MinHeap: child keys must be greater or the same.

Figure 5.1.: *Minimum heap property* in the BinaryHeap.

Each heap entry must be pointed to only once, with the exception of the dummy

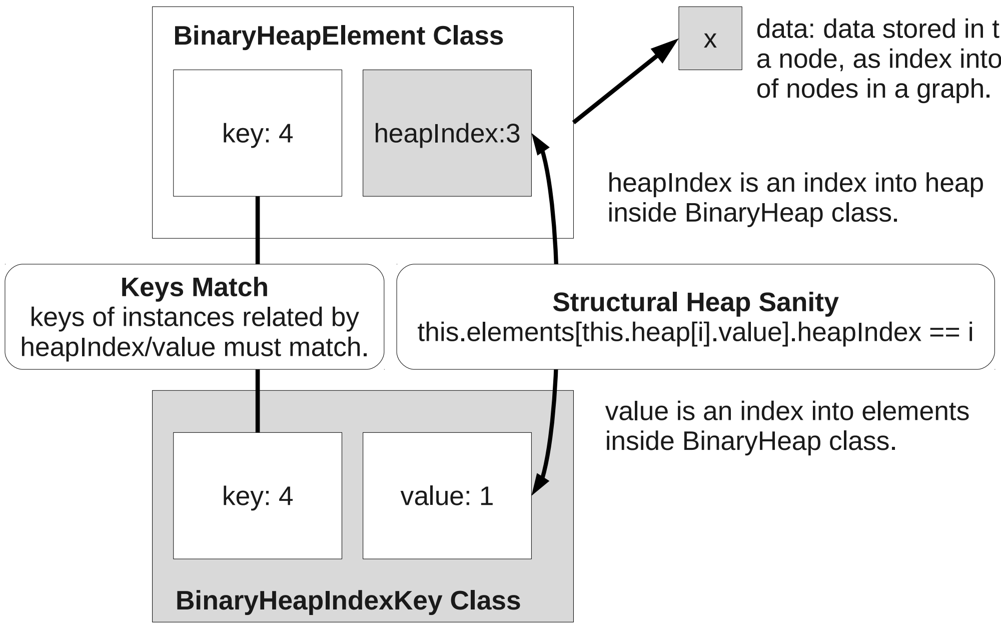

BinaryHeapElement Class
key: 4
heapIndex:3
x
data: data stored in the heap: a node, as index into an array of nodes in a graph.
heapIndex is an index into heap inside BinaryHeap class.
Keys Match
keys of instances related by heapIndex/value must match.
Structural Heap Sanity
this.elements[this.heap[i].value].heapIndex == i
value is an index into elements inside BinaryHeap class.
key: 4
value: 1
BinaryHeapIndexKey Class

Figure 5.2.: *Keys Match* and *Structural Heap Sanity* invariants in the BinaryHeap.

heap entry, where all elements removed from the heap point to. Different elements could be pointing to the same heap entry, causing problems when the heap gets restructured. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
// only dummy element can be pointed-to multiple times
/*@ invariant (
  @ (\forall int i, j;
  @  ((i >= 0) &&
  @   (j > i) &&
  @   (j < this.elementsLength) &&
  @   (i < this.elements.length) &&
  @   (j < this.elements.length))
  @   ==> (((this.elements[i] != null) &&
  @   (this.elements[j] != null) &&
  @   (this.elements[i].heapIndex ==
  @       this.elements[j].heapIndex))
  @   ==> (this.elements[i].heapIndex == 0)))
  @ ); @*/
```

Keys must not be negative. This is not an issue in the context of `FlatBaseDijkstra.run()`, but necessary in bottom-up mode when checking methods out of context.

```
// non-negative keys
/*@ invariant (
  @ (\forall int i;
  @    ((i >= 0) &&
  @    (i < this.heap.length) &&
  @    (i < this.heapLength))
  @    ==> ((this.heap[i] != null) &&
  @    (this.heap[i].key >= 0)))
  @ ); @*/
```

The individual BinaryHeap methods also have additional preconditions, mostly restrictions on the valid ranges for parameters.

- `BinaryHeap` constructor

  There are no preconditions for the `BinaryHeap` constructor. The constructor can be called at any time.

- `BinaryHeap.insert()` method

  There are two additional preconditions for the `BinaryHeap.insert()` method. The key of the inserted element must be valid and an overflow of the arrays must be avoided. The former is always the case in the context of `FlatBaseDijkstra.run()`, the latter is an implementation level issue caused by the Java port that does not use a flexible length vector.

```
/*@ requires (
  @ (key >= 0) &&
  @ (this.elementsLength - 1 < this.capacity)
  @ ); @*/
```

- `BinaryHeap.decreaseKey()` method

  There are some additional preconditions to the `BinaryHeap.decreaseKey()` method. The element in question must be valid, it must still be in the heap (not pointing to the dummy entry), and the new key must be valid and actually lower than the previous key of the element. These preconditions check that arguments are valid in a way always the case in the context of `FlatBaseDijkstra.run()`.

```
/*@ requires (
  @ (slotId >= 1) &&
  @ (slotId < this.elementsLength) &&
  @ (newKey >= 0) &&
  @ (newKey < this.elements[slotId].key) &&
  @ (slotId < this.elements.length) &&
  @ (this.elements[slotId] != null) &&
  @ (this.elements[slotId].heapIndex > 0) &&
  @ (this.elements[slotId].heapIndex < this.heapLength)
  @ ); @*/
```

- `BinaryHeap.min()` method

  The `BinaryHeap.min()` method only works if there is at least one element in the heap. This is a implementation level issue.

```
//@ requires (this.heapLength - 1 != 0);
```

- `BinaryHeap.minElement()` method

  The `BinaryHeap.minElement()` method only works if there is at least one element in the heap. This is a implementation level issue.

```
//@ requires (this.heapLength - 1 != 0);
```

- `BinaryHeap.deleteMin()` method

  The `BinaryHeap.deleteMin()` method only works if there is at least one element in the heap. This is a implementation level issue.

```
//@ requires (this.heapLength - 1 != 0);
```

- `BinaryHeap.clear()` method

  The BinaryHeap.clear() method works in all as long as the class invariants are respected in the prestate. No additional preconditions are required.

- `BinaryHeap.size()` method

  The `BinaryHeap.size()` method works in all as long as the class invariants are respected in the prestate. No additional preconditions are required.

- `BinaryHeap.empty()` method

  The `BinaryHeap.empty()` method works in all as long as the class invariants are respected in the prestate. No additional preconditions are required.

Richard A. Kemmerer suggests to integrate formal specification into the development process [Kem90]. That way information does not need to be reverse-engineered. Recovering lost information is much more difficult than saving it in the first place. We spent a lot of time creating preconditions the original developers must have known about to make thinks work at all. Moreover, the design could be influenced to make integrity checks easier.

5.2.3. Modifies Clauses

JForge requires that all data changed by the verified method must be explicitly listed in a `@Modifies` clause as shown in the example below:

```
@Requires(...
@Modifies("this.heap.elems, " +
"BinaryHeapElement.key, BinaryHeapElement.heapIndex, " +
"BinaryHeapIndexKey.key, BinaryHeapIndexKey.value")
@Ensures(...
public void decreaseKey(int slotId, int newKey) { ...
```

All other state is required to be the same in the prestate and in the poststate. An error will be reported otherwise.

In the top-down approach the `@Modifies` clause can be created by trial and error as the modified objects are not known a priori. For the `FlatBaseDijkstra.run()` method this are basically all objects. The input graph will remain unmodified, but as the implementation of Dijkstra's Algorithm tracks distances and state in the nodes themselves this is not completely true, either. The `@Modifies` clause may have an influence on the performance of the bounded program verification tool, but from a user's point of view there is no real meaning in the clause.

The clause is much more important in constraint mode, e.g in the bottom-up approach. Whenever a submethod is replaced by it's specification all state listed in the `@Modifies` clause is lost and the only thing guaranteed is that the specified postcondition holds. Take the `overtake()` method from the `race` example that swaps two elements in an arrays as an example. The specification only requires values for the two affected cars. If the array is marked as modifiable in constraint mode a counterexample might be generated where all other cars are set to null (or any arbitrary value) after calling this method, even though this certainly does not happen if executing the code. JForge assumes some state of the heap that is consistent with the specified postcondition. The prestate is irrelevant if the poststate does not refer to it via `@old`, and the code in the submethod is irrelevant as well. If a certain value is expected or deemed sensible for something it must be specified or not listed in the `@Modifies` clause.

In `BinaryHeap.insert()` the `heap` and `elements` arrays are listed in the `@Modifies` clause to allow the heap to be restructured. In constraint mode this means that all elements might be deleted from the heap or be assigned arbitrary values. While one might assume that state that is not referred to in the specification or the code does not change. This leads to counterexamples with traces that are not feasible as executions of the real code. It is the reason that the functional specification of BinaryHeap is not strict enough to use it in constraint mode.

5.3. Benchmarks and Scalability

With JForge runtime and memory consumption are an issue. The `FlatBaseDijkstra.run()` method consists of two nested loops. Unrolling three times instead of twice increases the unrolled code by a factor of 9/4, disregarding the fact that the loops in called methods are also unrolled more often, resulting in even worse slowdown. Consequentially the number of loop unrollings cannot be increased to high values.

Table 5.1 shows the JForge and InspectJ performance. All benchmarks were executed on the same machine with 8 2.66 GHz cores and 32 GB of RAM under openSUSE 11.3 (x86_64). CPU time and memory usage of the bounded program verification tools was determined using GNU time.

We can obtain the SAT formulas generated by JForge by replacing the Alloy library with a custom version. Using Z3 on the formulas instead of SAT4J speeds up JForge, but breaks postprocessing and does not solve the scalability issues. The listed numbers were obtained using SAT4J. With InspectJ performance is a lot more promising, but it became available too late for this thesis and has some usability issues discussed in chapter 5.5.

With JForge Java VM settings had to be tweaked as follows:

```
./eclipse -vmargs -Xmx22600m -Xms22600m \
    -XX:MaxPermSize=22600m -XX:-UseGCOverheadLimit
```

With InspectJ the following settings are required:

```
java -Xss100m -Xmx4G -Xms4G [...]
```

5.4. Specification Metrics

The amount of specification is difficult to measure, for similar reasons why counting lines of code is difficult. Moreover, as the specification is not complete, it is difficult to decide how much is missing. For an estimate see chapters 4.3.2, 4.4.1, and 5.2.2. It is obvious that a few high level annotations are not sufficient.

The original implementation typically runs on graphs with millions of nodes with an integer bitwidth of 32 or even 64. This requires millions of instances and millions of loop unrollings.

Table 5.1.: Benchmarking Results

Method	B	I	L	JF	stdev	JF mem	stdev	IJ	stdev	IJ mem	stdev	Result
decreaseKey	3	3	3	38.1	0.8	4631.4	7.6	5.1	0.1	973.9	17.4	unsat
	4	4	4	TO	TO	TO	TO	17.4	0.1	966.4	18.0	unsat
	5	5	5	—	—	—	—	25.0	0.2	978.3	8.7	unsat
	6	6	6	—	—	—	—	120.8	0.1	979.8	15.1	unsat
deleteMin	3	3	3	36.2	0.7	5159.5	17.5	5.1	0.0	997.2	13.5	unsat
	4	4	4	TO	TO	TO	TO	8.0	0.8	1013.7	11.6	unsat
	5	5	5	—	—	—	—	126.9	0.1	1017.7	6.0	unsat
insert	3	3	3	31.4	0.4	4624.0	119.4	5.1	0.1	984.5	15.6	unsat
	4	4	4	199.1	18.7	5938.1	47.6	11.6	3.3	975.7	10.3	unsat
	5	5	5	TO	TO	TO	TO	83.2	0.4	994.6	20.2	unsat
minElement	4	4	4	38.0	0.5	5999.6	55.4	4.2	0.0	954.6	3.7	unsat
	6	6	6	TO	TO	TO	TO	4.4	0.0	936.4	10.4	unsat
	8	8	8	—	—	—	—	12.9	0.1	955.5	0.8	unsat
	10	10	10	—	—	—	—	43.2	0.1	947.2	13.2	unsat
	11	11	11	—	—	—	—	315.2	2.0	957.7	3.9	unsat
run	3	3	1	50.0	3.7	11074.4	147.8	11.2	0.1	2963.2	15.0	sat
	4	4	1	235.5	40.8	28911.7	404.0	12.6	0.7	2910.3	17.6	sat
	7	1	1	TO	TO	TO	TO	5.1	0.1	1002.1	23.2	sat
	10	10	1	—	—	—	—	25.3	14.2	2934.9	22.1	sat
	3	3	2	TO	TO	TO	TO	344.1	435.5	4776.9	26.1	sat

Legend to Table 5.1

Column	Description	Unit
Method	name of the checked method	—
B	integer bitwidth	[1]
I	number of instances	[1]
L	number of loop unrollings	[1]
JF	mean JForge CPU time	[seconds], TO denotes timeout
JF mem	mean JForge memory consumption	[MB]
IJ	mean InspectJ CPU time	[seconds], TO denotes timeout
IJ mem	mean InspectJ memory consumption	[MB]
Result	counterexample found	(un)sat: (no) counterexample found
stdev	standard deviation of previous value	—

The scopes with JForge or even InspectJ are many orders of magnitude lower. As can be seen in table 5.1 not even a scope of 6-6-6 is possible for all methods.

The small scope hypothesis postulates that relatively small scopes are sufficient to discover most forgotten cases, off-by-ones and logical errors [ADK03]. Unfortunately there is no tool support to determine the required scope to achieve a certain amount of coverage. It is possible to check if certain lines in the code have been reached at all, by deliberately inserting bugs at certain places, however.

The following scopes are required to hit certain statements in the code:

Location	**Bitwidth**	**Instances**	**Loop Unrollings**
inner loop in `upheap()`	3	3	1
`break` in `deleteMin()`	4	4	1
inner `if` in `deleteMin()`	4	5	1

The scopes required to find the bugs mentioned in chapter 5.2.1 are listed in the table below. With smaller scopes the tools do not find the bugs and report unsat.

Bug	**Bitwidth**	**Instances**	**Loop Unrollings**
Memory Access Bug	3	2	1
First Aliasing Bug	4	4	1
Second Aliasing Bug	3	3	1

Statistics for running the code on simple graphs are listed in the table below.

Graph	**Bitwidth**	**Instances**	**Loop Unrollings**
1 Node, 0 Edges	4	4	1
2 Nodes, 1 Edge	4	5	2
5 Nodes, 20 Edges, Varying Weights	6	20	5

The number of instances and loop unrollings is determined by running instructed code that prints the relevant numbers. The bitwidth is the minimal number of bits required to store the number of instances in two's complement, typically required to iterate over an array containing all instances.

5.5. Usability of Bounded Program Verification Tools

JForge and InspectJ are specialized tools on the forefront of research, so they don't receive as much polish as more mainstream applications. There were problems with the feature sets and usability that slowed things down unnecessarily.

Some Java features are unsupported by both JForge and InspectJ:

- static methods †
- generics †
- Java Class Library is not supported fully †
- `Integer.MAX_VALUE` is not handled in a useful way †
- real arithmetic
- autoboxing
- multithreading

The features marked with † require more or less involving workarounds in the codebase or specification of Dijkstra's Algorithm. In other projects support for other features might be required.

The annotation syntax for the tools is not compatible, so two versions of the specification had to be maintained. Moreover, the annotations cannot be in the same file, so there exist two versions of the code.

Neither tool fully utilizes multiple cores resulting in bad performance on contemporary hardware. For unknown reasons both tools create formulas non-deterministically, making benchmarks difficult.

Postprocessing is weak with both tools: InspectJ does not explicitly list the prestate and the poststate, but displays the solution to the internal SMT formula with very little tracing information. JForge explicitly lists the prestate and poststate, but the mapping back to the Java source code is not perfect either. Overflows are not handled like exceptions. In the case of an uncaught exception or overflow neither tool links back to the Java source statement.

Compound statements are split up with JForge creating lots of temporary variables with cryptic names, making it difficult to scan through a long trace of autogenerated identifiers. InspectJ does not even support compound statements. Thus the following refactoring is required:

```
arr[base+off] = value + shift;
```

becomes

```
int idx = base + off;
int tmp = value + shift;
arr[idx] = tmp;
```

If a method violates an invariant in a class with many invariants JForge does not give the problematic invariant, but only the conjugation of all invariants. InspectJ only gives a satisfying assignment to the formula it created.

Neither tool supports an automatic indication of coverage, e.g. statement coverage.

JForge was one of the first tools of its kind, but there have been no updates since 2009, and some problems remain unsolved: There are problems with parameters aliasing attributes. The GUI is very helpful in the beginning but the lack of a command line interface becomes troublesome later, because automating tests for benchmarks is difficult. There is a bug with inline mode, effectively turning it into constraint mode in some cases where submethods are annotated with specification. To work around this issue specification of submethods needs to be commented out to enforce the expected behavior.

InspectJ is still a young project, it is very fast allowing top-down approaches, but it has not been tested as thoroughly as JForge, so some bugs are not yet resolved: Constructors cannot be verified. All methods have to be public for verification. Using boolean expressions in preconditions (like `requires(!this.decativated)`) was not supported. Invalid counterexamples were created, sometimes with inconsitent prestate like `arr[3] == 3` where `arr` is of type `int[2]`; initially InspectJ did not offer to check for contradictions in the preconditions. InspectJ calls an external tool, dot, for visualization. Support for dot had to be disabled because counterexamples from `FlatBaseDijkstra.run()` were to big to be processed within one hour. InspectJ created invalid SMT formuals. Z3 crashed. Comments within JML were problematic.

During the work on this thesis we ran benchmarks to compare InspectJ against JForge for a paper on InspectJ [LNT12]. I added the following features to InspectJ to improve usability and scalability:

- Support for running InspectJ on 32bit machines, previously only 64bit architectures were supported by InspectJ.
- Support to start InspectJ from command line in addition to running it from within Eclipse for easier modification of command line arguments.
- Speedup by using `StringBuilder` instead of `String` when creating large formulas.

6. Conclusion

The original goal of this thesis was to perform a top-down bounded program verification of a real-world implementation of Dijkstra's Algorithm with minimal, functional level specification. This goal could not be reached because of performance issues with JForge and too complex counterexamples with InspectJ. The fallback solution of bottom-up verification yielded several interesting results, but also failed to verify the functional correctness of the central `FlatBaseDijkstra.run()` method because of problems with constraint mode and many implicit assumptions in the code.

During bottom-up verification we achieved the following interesting results: We detected, fixed and reported a bug in the original implementation of Dijkstra's Algorithm. We recovered many implicit assumptions that constitute bugs from a verification point of view. We specified methods from the BinaryHeap class and proved adherence to the specification. We created specification for Dijkstra's Algorithm listed in chapter 4.3.1.1 (although we could not check it successfully). We benchmarked InspectJ against JForge. The results are published in a paper [LNT12]. We found and reported bugs in InspectJ itself and added some features.

Bounded program verification of real-world code is hard, especially if there are many undocumented implicit assumptions and the code is not written with verification in mind. Current bounded program verification tools require some more polish before mainstream adaption will happen. The iterated workflow, especially analysis of counterexamples, is too time consuming and lacks tool support.

Upcoming versions of InspectJ with better postprocessing support might enable top-down verification of Dijkstra's Algorithm.

7. Outlook and Future Work

Verifying the full `FlatBaseDijkstra.run()` method within a reasonable scope is an unsolved problem. It might be achievable soon as InspectJ is constantly improving.

Once the `run()` method has been verified for one-to-all it should be verified for one-to-one and different optimizations on the algorithmic level should be considered, like bidirectional search or arc-flags [Del09].

The recovered implicit assumptions might be incorporated in the original code to improve reliability, but the additional checks might decrease performance.

The tools, especially the postprocessing phase, could be polished and a set of standardized regression tests and benchmarks could be developed to help improve comparability between the available tools.

Bibliography

[ABB$^+$05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY tool," *Software and System Modeling*, vol. 4, pp. 32–54, 2005.

[ADK03] A. Andoni, D. Daniliuc, and S. Khurshid, "Evaluating the small scope hypothesis," Tech. Rep., 2003. [Online]. Available: http://sdg.csail.mit.edu/pubs/2002/SSH.pdf

[boo12] (2012, July) Boost homepage. [Online]. Available: www.boost.org/libs/graph/doc/dijkstra_shortest_paths.html

[BP10] D. L. Berre and A. Parrain, "The sat4j library, release 2.2," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010. [Online]. Available: http://jsat.ewi.tudelft.nl/content/sd/JSAT7_4_LeBerre.pdf

[Che03] J.-C. Chen, "Dijkstra's shortest path algorithm," *Journal of Formalized Mathematics*, vol. 15, 2003. [Online]. Available: http://mizar.org/JFM/Vol15/graphsp.html

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction To Algorithms.* MIT Press, 2001.

[Del09] D. Delling, "Engineering and augmenting route planning algorithms," Ph.D. dissertation, Universität Karlsruhe (TH), Fakultät für Informatik, 2009. [Online]. Available: http://i11www.ira.uka.de/extra/publications/d-earpa-09.pdf

[Den09] G. D. Dennis, "A relational framework for bounded program verification," Ph.D. dissertation, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.161.942

[Dij59] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959, 10.1007/BF01386390. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[dMB08] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008. [Online]. Available: http://research.microsoft.com/projects/z3/z3.pdf

[DY12] G. Dennis and K. Yessenov. (2012, February) Forge website. [Online]. Available: http://sdg.csail.mit.edu/forge/

[Jac02] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 256–290, April 2002. [Online]. Available: http://doi.acm.org/10.1145/505145.505149 [jah12] (2012, February) Jahob homepage. [Online]. Available: http://lara.epfl.ch/w/jahob_system

[Kem90] R. A. Kemmerer, “Integrating formal methods into the development process,” *IEEE Softw.*, vol. 7, pp. 37–50, September 1990. [Online]. Available: http://dx.doi.org/10.1109/52.57891

[Kla10] V. Klasen, “Verifying dijkstra’s algorithm with key,” Master’s thesis, Universität Koblenz-Landau, Campus Koblenz, 2010. [Online]. Available: http://kola.opus.hbz-nrw.de/volltexte/2010/528/

[KMB97] M. Kaufmann, J. S. Moore, and O. Boyer, “An industrial strength theorem prover for a logic based on common lisp,” *IEEE Transactions on Software Engineering*, vol. 23, pp. 203–213, 1997.

[LBR99] G. T. Leavens, A. L. Baker, and C. Ruby, “Jml: A notation for detailed design,” 1999.

[LNT12] T. Liu, M. Nagel, and M. Taghdiri, “Bounded program verification using an smt solver: A case study,” in *5th International Conference on Software Testing, Verification and Validation (ICST)*, April 2012.

[Mei11] S. Meinert, “Engineering data generators for robust experimental evaluations - planar graphs, artificial road networks, and traffic information -,” Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), Fakultät für Informatik, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025685

[miz12] (2012, February) Mizar homepage. [Online]. Available: http://www.mizar.org/

[MK07] R. Mange and J. Kuhn, “Verifying dijkstra’s algorithm in jahob,” 2007. [Online]. Available: http://lara.epfl.ch/w/_media/dijkstra.pdf?id=verifying_dijkstra_s_algorithm_in_jahob&cache=cache

[MZ05] J. S. Moore and Q. Zhang, “Proof pearl: Dijkstra’s shortest path algorithm verified with acl2,” in *18th International Conference on Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science.* Springer-Verlag, 2005.

[Sch11] P. H. Schmitt, “A computer-assisted proof of the bellman-ford lemma,” *Karlsruhe Reports in Informatics*, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022513

[Yes09] K. Yessenov, “A lightweight specification language for bounded program verification,” Master’s thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2009. [Online]. Available: http://hdl.handle.net/1721.1/53184

Appendix

A. Deutsche Zusammenfassung

In dieser Studienarbeit wenden wir Techniken der *Bounded Program Verification* auf eine geschwindigkeitsoptimierte Implementierung des Dijkstra-Algorithmus an.

Bounded Program Verification ist eine formale Methode zur Verifikation von Computerprogrammen innerhalb gewisser Grenzen. Sämtliche Eingaben bis zu einer benutzerdefinierten Maximalgröße werden dabei analysiert; über das Verhalten bei größeren Eingaben wird jedoch keinerlei Aussage getroffen. Der Dijkstra-Algorithmus dient der Berechnung kürzester Wege in Graphen und wird beispielsweise in der Routenplanung verwendet. Der Algorithmus nutzt eine Prioritätswarteschlange, die oftmals als binärer Heap realisiert wird.

Wir verwenden die beiden Java-Tools JForge und InspectJ zur Verifikation und portieren daher zunächst die in C++ geschriebene Implementierung des Dijkstra-Algorithmus nach Java. Wir erstellen außerdem eine formale Spezifikation des Algorithmus. In der Studienarbeit stellen wir dazu zwei unterschiedliche Ansätze vor: einen *Top-Down* Ansatz, bei dem eine globale Vor- und Nachbedingung verifiziert wird und der abgearbeitete Code als Black Box behandelt wird sowie einen *Bottom-Up* Ansatz, bei dem die aufgerufenen Untermethoden betrachtet und einzeln verifiziert werden.

Beim *Top-Down* Ansatz stoßen wir auf Performanceprobleme, aber durch den *Bottom-Up* Ansatz ist es möglich, wichtige Eigenschaften der zugrundeliegenden Heap-Datenstruktur zu beweisen. Wir finden und melden den Entwicklern einen Fehler in der Implementierung, welcher durch vorhergegangenes Testen nicht entdeckt wurde. Wir entdecken auch mehrere Fehler in unserer ungetesteten Java-Version des Programms, sowie zahlreiche implizite, undokumentierte Annahmen der Implementierung. Darüber hinaus führen wir einen Benchmark zum Vergleich von InspectJ mit JForge durch, entdecken und melden Fehler in InspectJ selbst und fügen einige neue Funktionen zu InspectJ hinzu.

Die größten Herausforderungen im Rahmen der Studienarbeit sind Probleme mit der Performance und dem Ressourcenverbrauch von JForge, die zeitaufwändige und wenig automatisierte Analyse der Gegenbeispiele, insbesondere mit InspectJ, sowie die vielen impliziten Annahmen im Quellcode—dadurch, dass es sich um hochoptimierten Code handelt, der ursprünglich nicht zur Verifikation vorgesehen war.

Es stellt sich außerdem heraus, dass neben der Spezifikation der eigentlichen Funktionalität für eine erfolgreiche Verifikation erforderlich ist, dass in der Spezifikation auch auf Details der Implementierungs-Ebene eingegangen wird, und dass eine Spezifikation erstellt wird, deren Notwendigkeit in der konkreten Funktionsweise der *Bounded Program Verification*-Tools begründet ist.

Die erfolgreiche Verifikation im *Top-Down* Ansatz ist noch ausstehend. Die gegenwärtig verfügbaren Programme zur *Bounded Program Verification* benötigen noch einen gewissen Feinschliff, bevor sie im großen Stil angewendet werden können. Zukünftige Versionen von InspectJ mit besserer Nachbearbeitung der Gegenbeispiele könnten die erfolgreiche Verifikation des Dijkstra-Algorithmus ermöglichen.

B. Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Karlsruhe, den 06. Juli 2012

(Michael Nagel)