FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Dijkstra's Algorithm---An (Almost) Verified Implementation
Student:	Jan Kupsa
Supervisor:	doc. RNDr. Dušan Knop, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

One thing is to write code that serves a certain purpose. Another thing is to have a code that reliably does only what it is supposed to do. One way to ensure this about your code is to use the annotation language ACSL and create what is usually called a verified code. ACSL allows us to specify the expected behavior of individual functions using annotations. These annotations, together with the code (written in the C programming language), can be processed by an external tool (FRAMA-C) and then formally proven within this tool that the code meets formal definitions for those functions.

Objectives of the work:
• Study Dijkstra's algorithm and describe it.
• Study the verification of C programs using the FRAMA-C framework and describe its basic usage.
• Implement Dijkstra's algorithm in the C language and propose the most comprehensive set of verifiable properties using FRAMA-C.
• Discuss which properties could not be verified and why this is not possible in the FRAMA-C framework.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 1 February 2024 in Prague.

Bachelor's thesis

DIJKSTRA'S ALGORITHM—AN (ALMOST) VERIFIED IMPLEMENTATION

Jan Kupsa

Faculty of Information Technology
Department of Information Security
Supervisor: doc. RNDr. Dušan Knop, Ph.D
May 16, 2024

Citation of this thesis: Kupsa Jan. *Dijkstra's Algorithm—An (Almost) Verified Implementation.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of code listings

I wish to thank doc. RNDr. Dušan Knop, Ph.D. for his help, patience, and advice in guiding me through writing this thesis.

Declaration

In Prague on May 16, 2024

Abstract

This thesis explores the application of formal verification to ensure the security and correctness of the implementation of Dijkstra's algorithm. The text is structured into seven chapters that will introduce the reader to graph theory, the shortest path problem, Hoare logic and the Frama-c framework for analysis of programs written in the C language. It will explain the internal plugins and their use in formal verification, as well as how to properly annotate source code using the ANSI/ISO Specification Language. The result of this thesis is a verified implementation Dijkstra's algorithm for solving the single source shortest path problem.

Keywords Frama-c, ACSL, formal verification, shortest path problem, verified implementation

Abstrakt

Tato práce se zabývá využitím formální verifikace pro zajištění bezpečnosti a správnosti implementace Dijkstrova algoritmu. Text je strukturován na sedm kapitol, které seznámí čtenáře s teorií grafů, problémem nejkratší cesty, Hoarovou logikou a frameworkem Frama-c, který slouží k analýze programů napsaných v jazyce C. Dále tento text vysvětlí využití interních pluginů na formální ověření a také jak správně přidat anotace ze specifikačního jazyka ANSI/ISO do zdrojového kódu. Výsledek této práce je ověřená implementace Dijkstrova algoritmu řešící problém nejkratší cesty z jednoho zdroje.

Klíčová slova Frama-c, ACSL, formální verifikace, problém nejkratší cesty, ověřená implementace

Introduction

Every year, society's reliance on digital technology grows. From critical infrastructure and financial systems to healthcare and everyday personal devices, software plays an integral role in modern life. As this dependency increases, so do the dangers stemming from flaws, failures, and vulnerabilities of software systems.

To mitigate these dangers, applications should be properly tested. However, traditional testing and debugging methods do not offer a guarantee of the absence of flaws. To obtain such a guarantee, we use a process called *formal verification*, which utilizes formal methods of mathematics to prove or disprove the correctness of a system regarding certain specifications.

One major approach to formal verification is deductive verification, which consists of generating *proof obligations* from a system and its specifications. These specifications are often written in formal specification languages, such as ANSI/ISO C Specification Language, which provides a way to describe preconditions, postcondition, invariants, and other properties that the software must satisfy. The truth of proof obligations then implies that the system conforms to its specifications. To prove them, we use automated theorem provers and satisfiability modulo theories solvers.

The goal of this thesis is to implement Dijkstra's algorithm and use deductive verification to prove the implementation's correctness regarding chosen specifications. The theoretical portion of this thesis is focused on an introduction to graph theory, the shortest path problem, and formal verification through the use of the `Frama-c` platform and ANSI/ISO C Specification Language.

The first chapter introduces graph theory and the shortest path problem. The second chapter focuses on Dijkstra's algorithm. The third chapter describes the verification environment `Frama-c`. The fourth chapter is a brief introduction to Hoare Logic. The fifth chapter describes the ANSI/ISO C Specification Language. The sixth chapter documents the implementation and verification of Dijsktra's algorithm. The seventh chapter discusses the role of formal verification in information security.

Chapter 1

Graph Theory and the Shortest Path Problem

This chapter begins with an introduction to graph theory, focusing on the definition of key terms necessary for formulating the SHORTEST PATH problem. Later, we discuss the problem itself, explore its different variations and the problem's overall importance in the field of computer science. The primary resources for this chapter are the lecture presentations from the course BI-AG1 and monograph, *Introduction to Algorithms* [1].

1.1 Graph theory and definitions

The first term that must be defined is the graph itself, the fundamental building block of graph theory.

▶ **Definition 1.1.1** (Undirected Graph)**.** An **undirected graph** (or a *graph*) is an ordered pair (V,E) where:

- V is a nonempty finite set of *nodes* or *vertices*.
- E is a set of *edges*.

An edge is an unordered pair of nodes $\{u,\ v\}$, where $u,\ v \in V$. The set of all possible edges for set V can be expressed as $\binom{V}{2}$. Therefore $E \subseteq \binom{V}{2}$.

See Figure 1.1 for examples of undirected graphs.

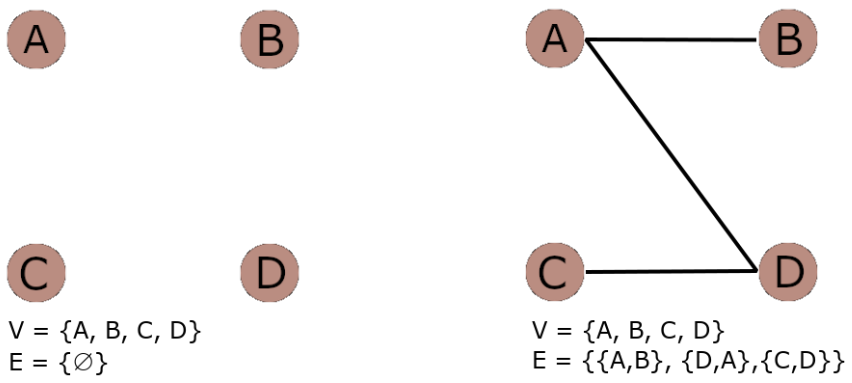


Figure 1.1 Examples of (undirected) graphs.

▶ **Definition 1.1.2** (Graph and edge terminology)**.** Let $e = \{u, v\}$ be an edge in an undirected graph $G = (V,E)$. We say:

- $V(G)$ and $E(G)$ denote the sets of all nodes and edges in graph G.
- nodes u and v are the **endpoints** of edge e.
- node u is **adjacent** to node v and vice versa.
- nodes u and v are **incident** with edge e.

▶ **Definition 1.1.3** (Walk and path in a graph)**.** Let G be a graph, then:

- **Walk of length** $k \geq 0$ **in the graph** G is a sequence of nodes and edges $v_0, e_1, v_1, e_2, ..., e_k, v_k$ where $e_i = \{v_{i-1}, v_i\}$, $e_i \in E(G)$ and $v_i \in V(G)$ for all $i = 1,...,k$.
- **Path in the graph** G is a walk with no repeated nodes and therefore no repeated edges.
- A path P with endpoints $s = v_0$ and $t = v_k$ is a **path from** s **to** t, alternatively an s-t-**path**.
- The **length** of any path P is defined as the number of edges in P.
- Let $s, t \in V(G)$ be two nodes in graph G. The node t is called **reachable from** s if there exists an s-t-path.

▶ **Definition 1.1.4** (Graph connectivity)**.** A graph $G = (V,E)$ is **connected** if there exists an s-t-path for each two nodes $s, t \in V(G)$. Otherwise, the graph is **disconnected**.

▶ **Definition 1.1.5** (Directed graph)**.** A **directed graph** is an ordered pair (V,E) where:

- V is a nonempty finite set of *nodes.*
- E is a set of oriented *edges.* $E \subseteq V \times V$

A directed edge is an **ordered** pair of nodes $(u,\ v)$ where $u,\ v \in V$. The node u is called the **predecessor** of v and v is the **successor** of u. An edge $(u,\ u)$ is called a **loop**.

See Figure 1.2 for examples of directed graphs.

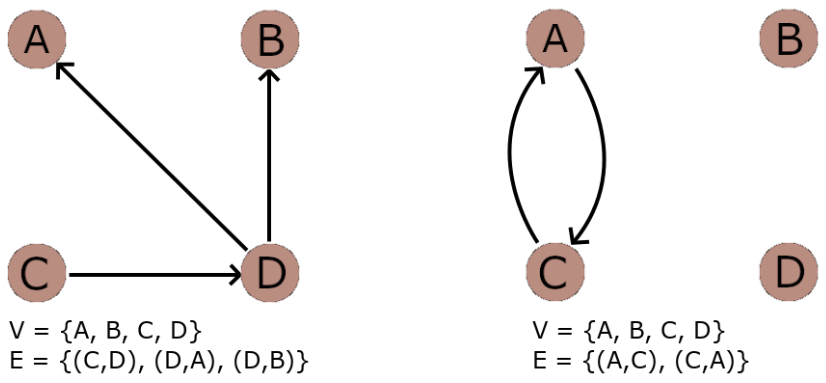


Figure 1.2 Examples of directed graphs.

▶ **Definition 1.1.6** (Subgraph)**.** Graph $H = (V,\ E)$ is a **subgraph** of graph $G = (V,\ E)$, if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

▶ **Definition 1.1.7** (Walk and path in a directed graph)**.** Let G be a directed graph, then:

- **Walk of length** $k \geq 0$ **in the graph** G is a sequence of nodes and directed edges $v_0, e_1, v_1, e_2, \ldots, e_k, v_k$ where $e_i = (v_{i-1}, v_i)$, $e_i \in E(G)$ and $v_i \in V(G)$ for all $i = 1,\ldots,k$.
- **Path in the graph** G is a walk with no repeated nodes.
- A path P with endpoints $s = v_0$ and $t = v_k$ is a **path from** s **to** t, alternatively s-t-**path**.
- The **length** of a path P is defined as the number of edges in P.
- Let $s,\ t \in V(G)$ be two nodes in graph G. The node t is called **reachable from** s if there exists an s-t-path.

▶ **Definition 1.1.8** (Tree)**.** Graph $G = (V,\ E)$ is called a **tree**, if there exists exactly one u-v-path for every pair of nodes $u,\ v \in V$.

▶ **Definition 1.1.9** (Symmetrization)**.** A **symmetrization** of a directed graph G is an undirected graph $\mathsf{sym}(G) = (V,\ F)$ where $V(G) = V(\mathsf{sym}(G))$ and $\{u,\ v\} \in F$ if either $(u,\ v) \in E$ or $(v,\ u) \in E$.

▶ **Definition 1.1.10** (Weak connectivity)**.** Let G be a directed graph, G is **weakly connected** if symmetrization $\mathsf{sym}(G)$ is connected. If graph G isn't weakly connected, it is **disconnected**.

▶ **Definition 1.1.11** (Strong connectivity)**.** A directed graph G is **strongly connected** if for every two nodes $u,v \in V(G)$ there exists a u-v-path and a v-u-path.

▶ **Definition 1.1.12** (Weighted graph)**.** Let $G = (V,\ E)$ be a connected undirected graph or a weakly connected directed graph, and let w a function $w\colon\ E \to \mathbb{R}$. After assigning a value $w(e)$ to each edge $e \in E$, the graph G becomes a **weighted graph**.

In weighted undirected graphs, the value of each edge is understood as the distance between the two endpoints. In weighted directed graphs, it represents the distance from the predecessor to the successor.

▶ **Definition 1.1.13** (Weight)**.** Let $G = (V,\ E)$ be a weighted directed graph with a weight function $w\colon\ E \to \mathbb{R}$ mapping edges to real-valued weights. The **weight** $w(P)$ of path $P = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_i \in V(G)$ for every $i \in \{0,\ 1,\ \ldots,\ \mathrm{k}\}$ is the sum of the weights of its edges: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

▶ **Definition 1.1.14** (Distance in weighted graphs)**.** The **distance** $d(u,v)$ between two nodes $u,\ v$ is the minimum among the weights of all u-v-paths; or $+\infty$, if there is no u-v-path. A **shortest path** from u to v is any path P with weight $w(P) = d(u,\ v)$.

Definitions 1.1.1 to 1.1.14 provide a way to describe several fundamental properties of both directed and undirected graphs. The key property explored in this thesis is the distance between any two given nodes u and v. To determine whether this u-v-path exists and find the shortest possible distance is commonly referred to as **the shortest path problem.**

1.2 The shortest path problem

A common variation of the problem poses the following task: given a graph G and two nodes $u,\ v \in V(G)$, find the shortest path from u to v.

The shortest path problem appears throughout society, which only serves to underscore its importance. In computer science, it is crucial for mapping software, data routing [2], and path-finding [3]. Logistics companies solve this problem to streamline delivery routes and maximize efficiency [4], while urban planners use it to design entire road networks and public transportation systems [5]. Given the problem's prevalence, it is natural that many algorithms have been proposed and developed to solve its many variations.

The problem given at the beginning of this section can also be referred to as the **single-pair shortest path problem**, signifying that the solution represents a path between two nodes only. Other variations include:

- The **single-source shortest path problem**: The solution provides the shortest path from a single node—called the source—to all other nodes reachable in the graph. A notable algorithm solving this task is the Breadth-first algorithm.
- The **all-pairs shortest path problem**: The goal is to find the shortest path between every possible pair of nodes in the graph.

1.2.1 Breadth-first search algorithm

The Breadth-first search algorithm (BFS for short) is a simple algorithm for, e.g., searching a graph, solving the single-source shortest path problem, and determining whether or not the given graph is connected. The proofs of the algorithm's properties are outside the scope of this thesis. The reason for including this algorithm is to familiarize the reader with the key idea upon which the algorithm is built. Many other algorithms expand on BFS: one such algorithm is Dijkstra's algorithm, which is the focus of this thesis.

Given a graph $G = (V,\ E)$ and a source node $v \in V(G)$, the algorithm systematically explores all edges in G, starting from the edges incident to v. It computes the distance from v to all other reachable nodes, where the distance is equal to the smallest number of edges between the source and the node. In doing so, it determines the length of the shortest path.

The name of the algorithm comes from the fundamental idea behind it. The algorithm 'spreads out' like a wave emanating from the source node, visiting the adjacent nodes first. Specifically, it marks all the nodes one edge away from the source v with a distance one. Then it marks all the nodes two edges away with a distance of two, and so on, terminating when the last node reachable from v has been marked.

The only data structure the algorithm uses is a first-in, first-out queue containing some nodes at a distance $k \in \mathbb{N}$, potentially followed by some nodes at a distance $k + 1$. This is because the algorithm evaluates an entire wave before it moves on.

To keep track of progress, the algorithm places each node in one of three states:

- **Undiscovered** nodes are those that have not been reached by a wave yet and are, therefore, still waiting to be put into the queue.

- **Discovered** nodes are those currently placed in the first-in, first-out queue.
- **Closed** nodes are the nodes that have been removed from the queue.

The algorithm also stores the **predecessor** of each node. Whenever it sets a node to the discovered state, it also marks the current node as the predecessor of the newly discovered node.

The pseudocode for the breadth-first-search procedure "BFS" on the following pages assumes that graph G is represented via adjacency lists. This simplifies the look-up of the neighbouring nodes at line 12. The algorithm also adds three attributes to each node in the graph:

- **State** holds the information about the node's state.
- **Distance** represents the wave in which the node has been discovered, thus the value is equal to the distance to the source.
- **Parent** is a pointer to the previous node.

The distance and parent attributes may be omitted, depending on the desired outcome of the algorithm. The distance attribute provides information about the length of each path discovered, whereas the parent attribute allows for the reconstruction of the full path from the source to any discovered node. Without these two attributes, the algorithm can still be used to determine if the given graph is connected.

The procedure works as follows. On lines 1–4 all nodes u are set to an undiscovered state. Their parent pointer gets set to NULL and the distance gets set to $+\infty$. Since the source node is always the first node that gets discovered, its attributes are set even before the main loop. On lines 5–7 the source node's state, distance and parent are set to discovered, 0 and NULL respectively. Lines 8–9 initialize the queue Q with the first and only node being the source node.

The **while** loop at 10–18 runs until the queue becomes empty, which will only happen when all discovered nodes get closed and there are no more reachable undiscovered nodes. The loop invariant is as follows: At line 10, the queue Q contains all nodes in the discovered state and no other nodes.

Line 11 removes the next node u to be processed from the queue Q. The **for** loop at lines 12–17 checks all the adjacent nodes v to node u and if any of them are undiscovered (line 13), then on lines 14–17 they become discovered. This is done by setting their state to discovered, setting v.distance to u.distance + 1, and marking u as v's parent in v.parent. On line 17 the newly discovered node gets added to the end of the queue Q.

Finally, after all the neighbouring nodes of the node u have been checked, u gets closed on line 18. Thus rendering all attempts at opening it again, on line 13, futile.

Code listing 1.1 BFS pseudocode

```
BFS(graph $G$, source node $s$)
    for each node $u \in V(G)$
        $u.state$ = undiscovered
        $u.distance$ = $+\infty$
        $u.parent$ = NULL
    $s.state$ = discovered
    $s.distance$ = 0
    $s.parent$ = NULL
    queue $Q$ = Empty Set
    ENQUEUE($Q$, $s$)
    while $Q$ is not empty
        $u$ = DEQUEUE($Q$)
        for each node $v$ adjacent to $u$
            if($v$.state == undiscovered)
                $v.state$ = discovered
                $v.distance$ = $u.distance$ + 1
                $v.parent$ = $u$
                ENQUEUE($Q$, $v$)
        $u.state$ = closed
```

The algorithm takes $O(|V|+|E|)$ time to complete for a graph $G = (V, E)$. The time complexity of initialization is $O(|V|)$. Each node can be enqueued and dequeued at most once. These two operations take $O(1)$ time and so the time spent on queue operations is $O(|V|)$. Each node's adjacency list gets scanned only when the node has been dequeued, therefore, it is scanned at most once. Given that the combined length of all adjacency lists is $\Theta(|E|)$, the total time dedicated to scanning the adjacency lists equals $O(|V| + |E|)$. Thus the total time complexity of the BFS procedure is $O(|V| + |E|)$.

1.2.2 Other notable algorithms

There are many algorithms designed to solve the shortest path problem, but as stated earlier, the problem is not uniform. Therefore, each algorithm has a set of preconditions that have to hold for the algorithm to be applicable.

Dijkstra's algorithm
This algorithm will be discussed in detail in Chapter 2.

Bellman–Ford algorithm
The Bellman-Ford algorithm solves the single-source shortest path problem for weighted directed graphs. It allows for some edges to have negative values. For the algorithm to solve the problem, however, it requires that the graph does not contain a negative cycle. Such a cycle makes it impossible to calculate the shortest path, as any path with a node on the cycle could be made shorter

by taking another walk around it. The Bellman-Ford algorithm is capable of detecting and reporting such a cycle.

It runs with $O(|V|^2 + |V| \cdot |E|)$ time complexity when the graph is represented by adjacency lists. This is because the algorithm finds the shortest paths in at most $|V|$ – *1* iterations and in each iteration, it scans all the edges. After the last iteration, it performs a final check of each edge, and if it finds a shorter path, it means there is a negative cycle.

A* search algorithm

The A* algorithm solves the single-pair shortest path for weighted graphs with non-negative weights. It is an informed search algorithm relying on a heuristic to speed up the solving process. The heuristic must be admissible for the specific graph being solved, such as the Manhattan or Euclidean distance in grid-based maps.

Due to the efficiency of its heueristic approach, the A* algorithm is used in various fields requiring optimal pathfinding solutions. Particularly, it is often used for AI pathfinding in games [6] and robotics [7]. The strength of this algorithm lies in the adaptability of the heueristic function, allowing for implementations that can work under specific constrains.

Floyd-Warshall algorithm

The Floyd-Warshall algorithm is dynamic-programming solution to the all-pair shortest-paths problem. It runs in $\Theta(|V|^3)$ time and is designed for weighted directed graphs. It allows for the existence of negative edges, but not negative cycles.

Johnson's algorithm for sparse graph

Johnson's algorithm finds the shortest path between all pairs of nodes in $O(|V|^2 \lg |V| + |V| \cdot |E|)$ time. It is designed to be asymptotically faster for sparse graphs (graphs with a low number of edges) than the Floyd-Warshall algorithm. It uses Dijkstra's algorithm and the Bellman-Ford algorithm as subroutines.

In order for the algorithm to use Dijkstra's algorithm, it has to perform a preprocessing of the given graph G to eliminate negative edges. This takes $O(|V| \cdot |E|)$ time.

The Bellman-Ford algorithm, apart from being used in the calculations, also provides Johnson's algorithm with the ability to detect and report a negative cycle.

See table 1.1 for a comparison of the discussed algorithms.

Name of the algorithm	Time complexity	Space complexity
Breadth-first search algorithm	$O(\|V\| + \|E\|)$	$O(\|V\|)$
Dijkstra's algorithm	$O(\|V\| \lg \|V\| + \|E\|)$	$O(\|V\|)$
Bellman-Ford algorithm	$O(\|V\|^2 + \|V\| \cdot \|E\|)$	$O(\|V\|)$
Floyd-Warshall algorithm	$\Theta(\|V\|^3)$	$O(\|V\|^2)$
Johnson's algorithm for sparse graph	$O(\|V\|^2 \lg \|V\| + \|V\| \cdot \|E\|)$	$O(\|V\|^2)$

Table 1.1 A list of algorithms for the SHORTEST PATH problem

Chapter 2

Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem, as described in 1.2, on a weighted directed graph with non-negative weights on all edges. The original version conceived by Edsger W. Dijkstra in 1956 solved only the single-pair shortest-path problem, but most variations today fix one node as the source.

The theory, procedure and proofs in this chapter are based on the monograph *Introduction to Algorithms*[1].

Dijkstra's algorithm can be thought of as a generalization of the breadth-first search algorithm to weighted graphs. A wave spreads out from the source, but the time it takes to reach a node is given by the weight of the edge it travels over, as opposed to taking a uniform unit of time as in breadth-first search.

That algorithm always selects the node with the shortest distance, but the shortest path to that node might not have the fewest edges. This means that the discovered node with the shortest distance might have been added after other discovered nodes. Therefore, a simple first-in, first-out queue will not suffice for selecting the node from which the next wave goes out.

Instead, the algorithm maintains a set S of all nodes with their final path lengths from the source determined. The algorithm then repeatedly selects a node u with the lowest shortest-path estimate from among the remaining nodes: $u \in V \setminus S$. It adds this node to the set S and **relaxes** all edges leaving u.

2.1 Relaxation

Relaxation is a technique used in multiple single-source shortest paths algorithms, including Dijkstra's algorithm. It is used to reevaluate the attributes $v.d$ and $v.parent$, both maintained for each node $v \in V$ by the algorithm. The attribute $v.d$ is called the **shortest-path estimate**. It represents the length of the shortest path from the source found at any moment during runtime.

Thus the value of the attribute can only decrease as shorter paths are found. The attribute *v.parent* stores the **predecessor** of the node v.

Relaxation first requires the initialization of both attributes for each node using the procedure INITIALIZE-SINGLE-SOURCE 2.1. This procedure runs in $\Theta(V)$ time and ensures that the source node $s \in V$ has its attributes set to $s.d = 0$ and $s.parent =$ NULL. Each node $v \in V \setminus \{s\}$ has its attributes set to $v.d = \infty$ and $v.parent =$ NULL.

Code listing 2.1 Initialize-Single-Source pseudocode

```
INITIALIZE-SINGLE-SOURCE(graph $G$, source node $s$)
    for each node $v \in V(G)$
        $v.d = +\infty$
        $v.parent$ = NULL
    $s.d = 0$
```

The process of **relaxing** an edge (u,v) where $u, v \in V(G)$ consists of calculating whether going through the edge to node v improves the shortest path to v found so far. If so, update $v.d$ and *v.parent*. The RELAX 2.2 procedure runs in $O(1)$ time. Figure 2.1 shows two examples of relaxing an edge, one which decreases the shortest-path estimate and one in which nothing changes.

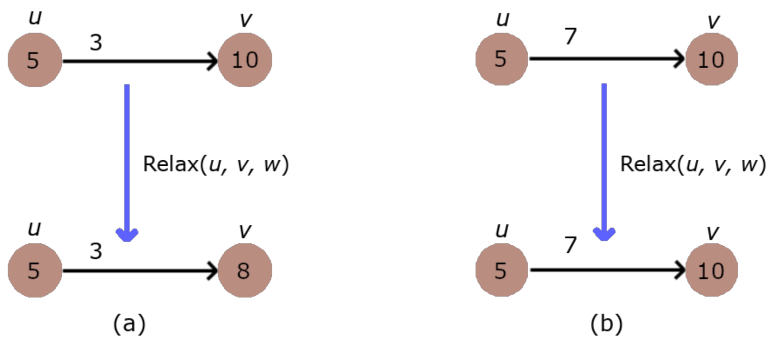


Figure 2.1 Relaxing two different edges, one with $w(u, v) = 3$ and one with $w(u, v) = 7$. The shortest-path estimate appears over each node. **(a)** The estimate changes because the inequality $u.d < v.d - w(u, v)$ holds prior to relaxation. **(b)** Because prior to relaxation, the inequality $u.d < v.d - w(u, v)$ does not hold, the relaxation step leaves the attributes unchanged.

Code listing 2.2 Relax pseudocode

```
RELAX(node $u$, node $v$, edge weight $w$)
    if $u.d < v.d - w(u, v)$
        $v.d = u.d + w(u, v)$
        $v.parent = u$
```

Relaxation is the only means by which the shortest-path estimate and predecessor attributes can change. Dijkstra's algorithm relaxes each edge exactly once.

2.2 Properties of Shortest Paths

To prove the theorems in the next section, we establish several properties of shortest-path estimates.

▶ **Lemma 2.1** (Upper-bound property)**.** *For all nodes $v \in V$ and a source node $s \in V$, the inequality $v.d \geq d(s, v)$ is always true. Once $v.d = d(s, v)$, the shortest-path estimate $v.d$ never changes.*

▶ **Corollary 2.2** (No-path property)**.** *If there is no path from the source s to node v, the shortest-path estimate $v.d = d(s, v) = \infty$.*

▶ **Lemma 2.3** (Convergence property)**.** *If s-u-path extended to node v by adding the edge (u, v), is a shortest path in G for some $u, v \in V(G)$, and if $u.d = d(s, u)$ before relaxing the edge(u, v), then $v.d = d(s, v)$ at all times after relaxing the edge (u, v).*

▶ **Lemma 2.4** (Predecessor-subgraph property)**.** *Once $v.d = d(s, v)$ for all nodes $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.*

2.3 Dijkstra procedure

The procedure maintains a set S of nodes whose final shortest-path estimate has already been determined. The algorithm then repeatedly selects a node u whose shortest-path estimate is the lowest among the remaining nodes: $u \in V \setminus S$. The procedure stores these remaining nodes in a min-priority queue Q, keyed by the value of their d attribute.

Code listing 2.3 Dijkstra's algorithm pseudocode

```
Dijkstra(graph G, source node s, edge weights w)
    Initialize-Single-Source(G, s)
    S = Empty Set
    Q = Empty Queue
    for each node u ∈ V(G)
        INSERT(Q, u)
    while Q is not empty
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each node v in G.Ajd[u]
            RELAX(u, v, w)
            if the call of Relax decreased v.d
                Decrease-Key(Q, v, v.d)
```

Line 1 initializes d and *parent* attributes and line 2 initializes the set S to an empty set. Lines 3–5 initialize the min-priority queue Q to contain all nodes in $V(G)$. The algorithm maintains the invariant $Q = V - S$ at the start of each **while** loop iteration on lines 6–12. The invariant is maintained because in every iteration, a node u is extracted from Q on line 7 and added to S on line 8. As such, vertex u has the smallest shortest-path estimate out of any node in $V - S$. Then, lines 9–12 relax each node v incident to node u, updating the $v.parent$ and decreasing $v.d$ if the path going through u is shorter than the path found so far, or if it is the first path found.

The min-priority queue Q is increased exclusively on lines 4–5. During the **while** loop, the number of nodes in the queue only decreases. This decrease, on line 7, is guaranteed to happen during each iteration, therefore, there are exactly $|V|$ iterations of the 6–12 **while** loop.

The selection of the node u on line 7 is based on its lowest shortest-path estimate. This can be thought of as a greedy strategy. Such strategies do not always result in optimal solutions, but in the case of Dijkstra's algorithm, it does lead to computing the shortest path. The key is to prove that $u.d = d(s, u)$ each time the node u is added to set S.

▶ **Theorem 2.5** (Correctness of Dijkstra's algorithm)**.**
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source node s, terminates with $u.d = d(s, u)$ for all nodes $u \in V(G)$.

Proof. To prove Theorem 2.5 we will show that $v.d = d(s, v)$ holds for every node $v \in S$ at the start of each iteration of the **while** loop on lines 6–12. This is enough because the algorithm terminates when $S = V$.

The proof is by induction on the number of iterations of the **while** loop, which is equal to $|S|$ at the start of every iteration. There are two initial cases: for $|S| = 0$, so that $S = \emptyset$ and the claim is true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = d(s,s) = 0$.

The induction hypothesis, for the induction step, is that $v.d = d(s, v)$ for all $v \in S$. After extracting the node u on line 7, and adding it to S on line 8, the shortest-path estimate $u.d$ never changes. Therefore we need to show that $u.d = d(s, u)$ on lines 7–8. First, if there is no path from s to u, then, by the no-path property, we are done. Second, if there is an s-u-path, then let y be the first node on the shortest s-u-path that is not in S, and let x be its predecessor on the shortest path. Because all weights are non-negative and y comes before u, we have $d(s, y) \leq d(s, u)$. Because the call of Extract-Min on line 7 returned node u as having the minimal shortest-path estimate $u.d$ of all nodes in $V - S$, we also have $u.d \leq y.d$. The upper-bound property gives us $d(s, u) \leq u.d$.

Because $x \in S$, the induction hypothesis states that $x.d = d(s, x)$. During the iteration of the **while** loop that added x to S, all edges outgoing from x, including the edge (x, y) were relaxed. By the convergence property, the node y was assigned the value $d(s,\ y)$ as its shortest-path estimate at that time. Therefore, we have $d(s,\ y) \leq d(s,\ u) \leq u.d \leq y.d$ and $y.d = d(s,\ y)$. That gives us $d(s,\ y) = d(s,\ u) = u.d = y.d$, hence $u.d = d(s,\ u)$, and by the upper-bound property, this value never changes again. ☐

▶ **Corollary 2.6.** *After Dijkstra's algorithm is run on a weighted, directed graph $G = (V,\ E)$ with nonnegative weight function w and source node s, the predecessor subgraph G_{parent} is a shortest-paths tree rooted at s.*

Proof. Theorem 2.5 establishes, that after Dijkstra's algorithm terminates, the equality $v.d = d(s,\ v)$ holds for all nodes $v \in V$. Therefore, the corollary is proved by the predecessor-subgraph property. ☐

Analysis

The time complexity of Dijkstra's algorithm depends on the implementation of the min-priority queue Q. To maintain this queue, the algorithm performs three distinct operations: INSERT on line 5, EXTRACT-MIN on line 7, and DECREASE-KEY on line 12. Both INSERT and EXTRACT-MIN are called exactly once for each node, while DECREASE-KEY is called at most once for every edge.

A simple implementation stores each shortest-path estimate in an array, where the index of each estimate corresponds to a node. The INSERT and DECREASE-KEY operations take $O(1)$ time and each EXTRACT-MIN takes $O(|V|)$ time since it has to search through the entire array. This gives a total time of $O(|V|^2 + |E|) = O(|V|^2)$.

A more efficient implementation uses a Fibonacci heap [8], which can improve the time to $O(|V| \lg |V| + |E|)$. This improvement is because the Fibonacci heap decreases the amortized cost of each of the $|V|$ EXTRACT-MIN operations to $O(\lg |V|)$ while keeping the amortized cost of every DECREASE-KEY operation at $O(1)$ time.

It is worth noting, that this procedure is best suited for connected graphs, as there are iterations of the **while** loop on lines 6–12 even for unreachable nodes. To avoid unnecessary calculations, the graph can be preprocessed with an algorithm like BFS in Section 1.2.1 to create a subgraph containing only reachable nodes.

In practice, the **while** loop would be adjusted to terminate once a node u with an estimate equal to the initial value, often INT_MAX, was extracted. A different approach is to introduce a state attribute to each node, similar to BFS, to keep track and select from only discovered nodes. However, such adjustments can lead to a loss of certainty as to when the algorithm will terminate. This certainty plays an important part in the verification of loop variants, as will be discussed in future chapters.

Chapter 3

Verification environment Frama-c

3.1 Frama-c

Frama-c (*Framework for Modular Analysis of C programs*) is a platform built for the analysis of source code written in C. By combining several analysis techniques into a single framework, Frama-c allows analyzers to use, and expand upon, the results already computed by other analyzers.

Frama-c can serve as a tool for several purposes, such as code analysis or lightweight semantic extraction. However, the focus of this thesis is to use Frama-c for formal verification through the use of ACSL (*ASNI/ISO C Specification Language*) 5 specifications.

3.2 Installation

Frama-c is distributed as source code, which includes the Frama-c kernel and a base set of open-source plug-ins used in the verification process. It is available on Linux, Mac, and Windows through the use of WSL (Windows Subsystem for Linux).

In this thesis, the operating system used is a Linux distribution Ubuntu 18.04, while Frama-c (v27.1 Cobalt) was installed via the recommended method using the OCaml package manager [9], opam.

First, install opam by running the following script:

```
bash -c "sh <(curl -fsSL
  https://raw.githubusercontent.com/ocaml/opam/master/shell/install.sh)"
```

After installation, opam needs to be initialized and a new *opam switch* has to be created.

```
    opam init
    opam switch create 4.11.2
    eval $(opam env)
```

Initialization of the *opam switch* takes time and disk space because it downloads and builds an OCaml compiler. After installing and initializing `opam`, run the following command to obtain the command-line `frama-c` executable; the graphical interface `frama-c-gui`; and `ivette`, the newest GUI.

```
opam install frama-c
```

Note that Ivette's dependencies are not included in `opam`, however, they are automatically downloaded from `npm` when `ivette` is run for the first time.

`Frama-c` version 27.1 comes with the external prover Alt-Ergo. Any additional provers, such as Z3 [10], CVC4 [11], Gappa [12], and PVS [13], can be installed at any time.

3.3 Plugins and Solvers

The `Frama-c` platform is designed to support two types of plug-ins: *external* and *internal*. These types differ in their distribution. The internal plug-ins are distributed within the `Frama-c` kernel, whereas the external are distributed independently.

These plug-ins expand the framework's capabilities. In this thesis, we will focus on plug-ins designed for formal verification.

3.3.1 Plugin: WP

This internal plugin is an implementation of the *Weakest Precondition* calculus and computes proof obligations of programs annotated with ACSL annotations. It is designed to not only use its own prover `Qed`, but also external automated provers such as Alt-Ergo or Z3-solver [10] through the use of the `Why3` platform. Furthermore, it is capable of using interactive proof assistants such as `Coq` [14].

`Frama-c/WP` use the following heuristic for discharging proof obligations:

1. try internal prover `Qed`,
2. try any SMT [15] prover,
3. try the `Coq` interactive proof assistant,
4. try any Tactic alternative,
5. try any remaining prover alternatives.

The WP plug-in is complementary to an older plug-in named Jessie, which is also an implementation of the weakest precondition calculus. Unlike Jessie, WP uses three different memory models: Hoare model, Typed model, and Bytes model. It also allows the user to combine the weakest precondition calculus with other techniques, like the EVA plug-in.

3.3.2 Plugin: EVA - Evolved Value Analysis

The internal plug-in EVA is founded on *Abstract Interpretation*. This method allows EVA to approximate the sets of values that variables might hold. With sufficient precision, the plug-in is capable of detecting potential runtime errors, such as out-of-bounds accesses, divisions by zero, and uses of uninitialized variables. The precision is set by the user through the use -EVA-PRECISION option. The allowed values are 0–11. This option offers a trade-off between precision and analysis time. A lower setting can lead to false positives, so it is important to find a proper precision.

3.3.3 Plugin: RTE - Runtime Error Annotation Generation

RTE is an internal plug-in that analyzes the source code and generates additional ACSL annotations for common runtime errors.

RTE is the most valuable in a modular setting, where its main purpose is to provide supplementary proof obligations to more advanced plug-ins, such as WP.

3.3.4 Solver: Alt-Ergo

Alt-Ergo is an open-source automatic solver of mathematical formulas, designed for program verification. It is based on *Satisfiability Modulo Theories*, SMT for short.

Originally developed at LRI (*Laboratoire de Recherche en Informatique*) [16] for the Why3 [17] platform, Alt-Ergo is now maintained and further developed by OCamlPro [18], in collaboration with Why3 [17]. Its use has also expanded; Alt-Ergo is now employed in SPARK [19] to verify formulas produced from Ada programs, in cryptographic protocols verification, and in B methods [20].

According to OCamlPro [21], Alt-Ergo is currently capable of reasoning in the combination of the following theories:

- the free theory of equality with uninterpreted symbols,
- linear arithmetic over integers and rationals,
- fragments of non-linear arithmetic,

- polymorphic functional arrays with extensionality,
- enumerated datatypes,
- record datatypes,
- associative and commutative (AC) symbols,
- fixed-size bit-vectors with concatenation and extraction operators.

3.3.5 Solver: Z3

`Z3`, first released in September 2007, [10] is a powerful SMT solver developed by the *Research in Software Engineering* group at Microsoft Research. `Z3` has been released as open source in 2015 and is now available on GitHub [22].

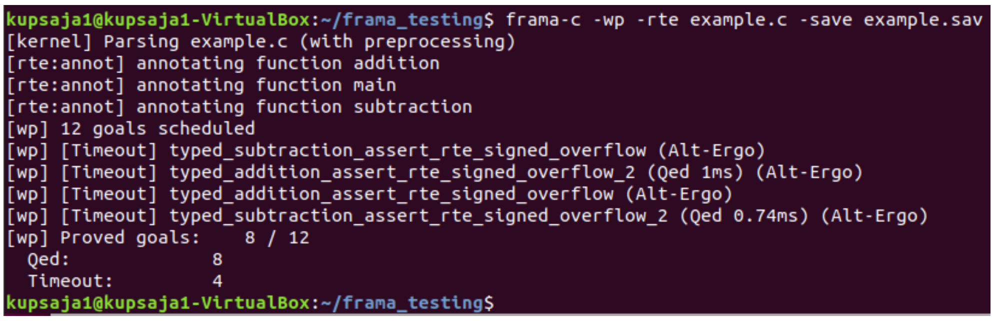
It supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, un-interpreted functions and quantifiers. `Z3` is used in a wide range of software engineering applications, including program verification, compiler validation, model-based software development, and network verification.

3.4 Example of Using Frama-c

To check whether the installation was successful and show how to use `Frama-c`, we will attempt to verify the program in Code listing 3.1. At first glance, the source code contains peculiar comments on lines 2, 6, and 10. These are ACSL notations and they will be explained in Chapter 5, for now, we won't focus on their meaning.

Code listing 3.1 Source code of an example program caption

```
1  //@ assigns \nothing;
2  int addition (int x, int y){
3        return x + y;
4  }
5  //@ assigns \nothing;
6  int subtraction (int x, int y) {
7      return x - y;
8  }
9  //@ assigns \nothing;
10 int main(){
11     int x = 2147483646;
12     int y = 3;
13     int result = subtraction(x,y);
14     result = addition(x,y);
15     return 0;
16 }
```



```
kupsaja1@kupsaja1-VirtualBox:~/frama_testing$ frama-c -wp -rte example.c -save example.sav
[kernel] Parsing example.c (with preprocessing)
[rte:annot] annotating function addition
[rte:annot] annotating function main
[rte:annot] annotating function subtraction
[wp] 12 goals scheduled
[wp] [Timeout] typed_subtraction_assert_rte_signed_overflow (Alt-Ergo)
[wp] [Timeout] typed_addition_assert_rte_signed_overflow_2 (Qed 1ms) (Alt-Ergo)
[wp] [Timeout] typed_addition_assert_rte_signed_overflow (Alt-Ergo)
[wp] [Timeout] typed_subtraction_assert_rte_signed_overflow_2 (Qed 0.74ms) (Alt-Ergo)
[wp] Proved goals:    8 / 12
  Qed:             8
  Timeout:         4
kupsaja1@kupsaja1-VirtualBox:~/frama_testing$
```

Figure 3.1 Result of the verification of the example program in Code listing 3.1.

Verification from the command line

To verify the program using the Frama-c framework, we will use the command FRAMA-C in a terminal. This command accepts several options: the specific plug-ins to be used, the name of the file containing the source code, and the -SAVE option followed by a filename where the result of the analysis, and the current session, will be stored.

```
frama-c -wp -rte example.c -save example.sav
```

If everything was installed and configured properly, the syntax of both ACSL and C was followed, then Frama-c completes the analysis and provides the results that can be seen in Figure 3.2.

To display the results again in the future, or continue with the session, we can use the option -LOAD followed by a valid filename (in this case EXAMPLE.SAV), and potentially other plug-ins which hadn't been used yet.

```
frama-c -load example.sav -eva
```

If we hadn't used the -SAVE option, the session would end upon completing the command. The contents of the .SAV files are not human readable, to view them, the commands FRAMA-C-GUI or ivette are necessary.

Output analysis

In the output 3.1, it is clear that Frama-c first used RTE plug-in to add new ACSL annotations to functions ADDITION, MAIN, and SUBTRACTION. The WP plug-in was then scheduled to prove 12 goals. It succeeded at proving 8 of them, using WP's built-in solver Qed. The remaining 4 were judged to be too complex for Qed and were handed off to the more capable Alt-Ergo. However, Alt-Ergo reached a timeout limit, resulting in an unknown status for these 4 goals.

Timeout setting

The timeout limit can be set manually, using the -WP-TIMEOUT option followed by the maximum allowed time in seconds.

```
frama-c -wp -wp-timeout 5 -rte example.c
```

Keep in mind that this limit will be applied to every goal scheduled, which can drastically prolong the verification time. Therefore, it might be beneficial to split the code into multiple header and source files and verify them independently.

Solver specification

Because no solver was specified, `Frama-c` defaulted to connect `Alt-Ergo` to `WP`. `Frama-c` is capable of using multiple solvers at once. To select them, use the option -WP-PROVER followed by a list of solvers, separated by a comma.

```
frama-c -wp -wp-prover z3,alt-ergo -rte example.c
```

Analysis improvement

Notice that the four failed goals shown in Figure 3.1 all contain "rte", that is because the annotations, that led to the scheduling of these goals, were created by the `RTE` plug-in. To come to any conclusion about these goals, an additional plug-in—one specialised in run-time errors—is required. We will run the command again, but this time with the `EVA` plug-in.

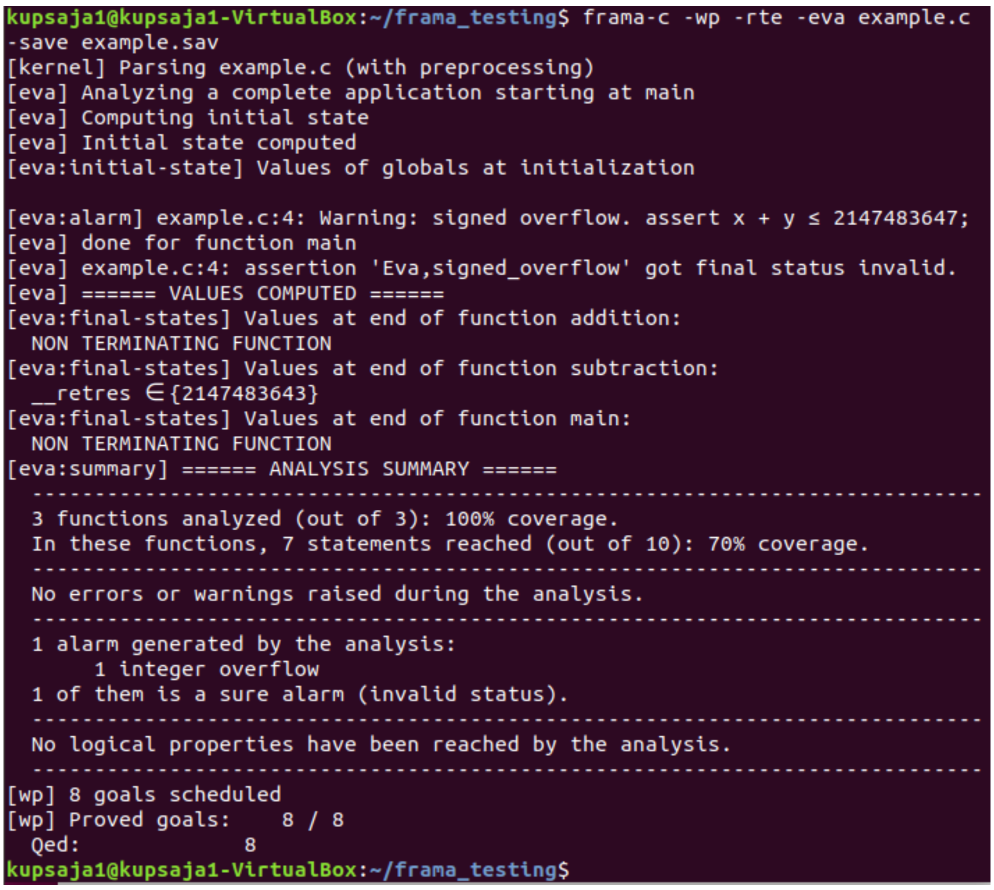
```
frama-c -wp -rte -eva example.c -save example.sav
```

The outcome shown in Figure 3.2 of the analysis is now completely different. Every line, except the first and the last 3, is an output from the `EVA` plugin.

`EVA` structures its output into three categories: the analysis, the values computed, and the analysis summary. In the first part, we can see that `EVA` raised an alarm about a potential signed overflow and created an ACSL *assert* notation. Two lines later it evaluated the assertion as *invalid* and stopped the analysis. Invalid annotations indicate that the code is confirmed to behave in opposition to the behaviour specified with the ACSL notation.

The premature end to the analysis can also be seen in the third part, the analysis summary. It states that out of all the functions analyzed, only 70% of statements have been reached. This indicates that `EVA` has either stopped analysing, or it calculated that some part of the code will never be executed; perhaps due to variables never satisfying an IF statement.

As mentioned previously, `EVA` may report false positives. By increasing the precision through -EVA-PRECISION, these alarms can be eliminated. However, in this example, increasing precision would not achieve anything, because all variables are set and never change. Therefore, `EVA` knows the exact values of x and y at any given moment.



```
kupsaja1@kupsaja1-VirtualBox:~/frama_testing$ frama-c -wp -rte -eva example.c
-save example.sav
[kernel] Parsing example.c (with preprocessing)
[eva] Analyzing a complete application starting at main
[eva] Computing initial state
[eva] Initial state computed
[eva:initial-state] Values of globals at initialization

[eva:alarm] example.c:4: Warning: signed overflow. assert x + y ≤ 2147483647;
[eva] done for function main
[eva] example.c:4: assertion 'Eva,signed_overflow' got final status invalid.
[eva] ====== VALUES COMPUTED ======
[eva:final-states] Values at end of function addition:
  NON TERMINATING FUNCTION
[eva:final-states] Values at end of function subtraction:
  __retres ∈{2147483643}
[eva:final-states] Values at end of function main:
  NON TERMINATING FUNCTION
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------------------
  3 functions analyzed (out of 3): 100% coverage.
  In these functions, 7 statements reached (out of 10): 70% coverage.
  ----------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------------------
  1 alarm generated by the analysis:
       1 integer overflow
  1 of them is a sure alarm (invalid status).
  ----------------------------------------------------------------------------
  No logical properties have been reached by the analysis.
  ----------------------------------------------------------------------------
[wp] 8 goals scheduled
[wp] Proved goals:    8 / 8
  Qed:             8
kupsaja1@kupsaja1-VirtualBox:~/frama_testing$
```

Figure 3.2 Result of the verification of the example program 3.1 using `EVA`.

The last thing to note is the complete success of the `WP` plug-in. In this case, `EVA` has handled all annotations regarding runtime, leaving `WP` with only the goals it is capable of proving.

Graphical interface of `Frama-c`

To use the graphical interface, we only change the command from FRAMA-C to FRAMA-C-GUI.

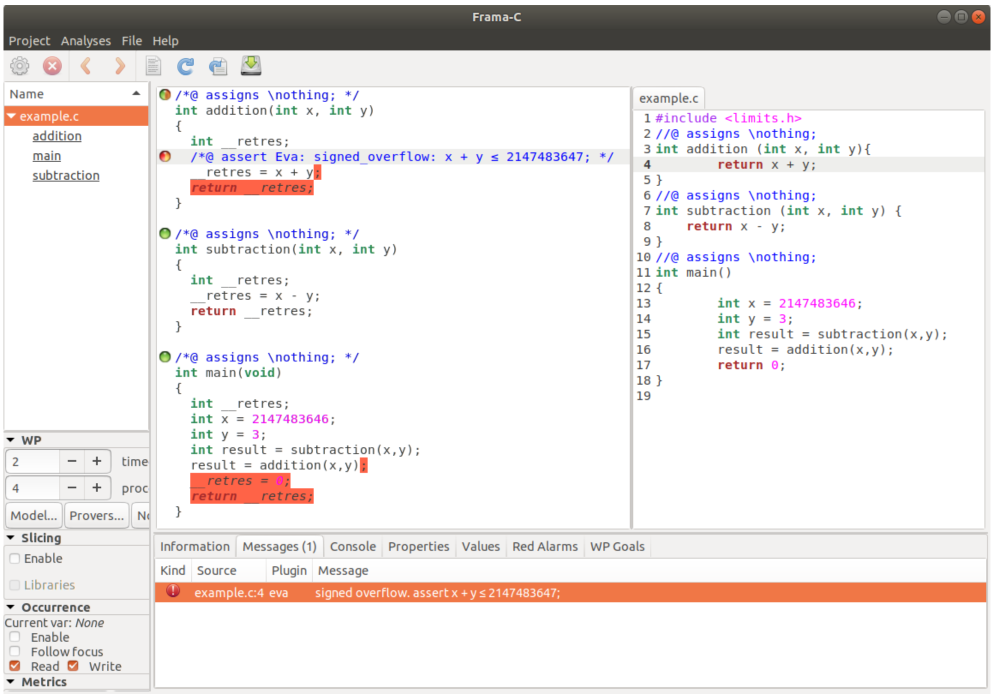
```
frama-c-gui -wp -rte -eva example.c -save example.sav
```

The graphical interface, as seen in Figure 3.3, has five important parts:

- **Plug-in view:** In the bottom left corner is a menu of options for each plug-in and other capabilities of the framework. Adjusting these options, coupled with the -SAVE and -LOAD commands, can help in fine-tuning the verification process.
- **File tree:** This tab shows the structure of the source code files. It divides them into functions and specific ACSL structures such as predicates.

Selecting different parts changes the main view in the middle.

- **Normalized source code view:** The largest window in the middle shows the selected part of the file tree. Clicking on any line will display the corresponding information in the ”Information” tab of the messages view.
- **Original source code view:** In this view, we can see the original source code.
- **Messages view:** This view is comprised of several tabs:
  - **Information:** displays a brief report on the currently selected object.
  - **Messages:** shows messages generated by the `Frama-c` kernel and plug-ins. It also shows all raised alarms.
  - **Console:** Contains the same output as if the FRAMA-C command was used.
  - **Properties:** shows the statuses of properties.
  - **Values:** displays information pertaining to the `EVA` plug-in.
  - **Red Alarms:** displays properties evaluated as *invalid* or *invalid under hypotheses.*
  - **WP Goals:** displays information related to the `WP` plug-in.



```
/*@ assigns \nothing; */
int addition(int x, int y)
{
  int __retres;
  /*@ assert Eva: signed_overflow: x + y ≤ 2147483647; */
  __retres = x + y;
  return __retres;
}

/*@ assigns \nothing; */
int subtraction(int x, int y)
{
  int __retres;
  __retres = x - y;
  return __retres;
}

/*@ assigns \nothing; */
int main(void)
{
  int __retres;
  int x = 2147483646;
  int y = 3;
  int result = subtraction(x,y);
  result = addition(x,y);
  __retres = 0;
  return __retres;
}
```

```
#include <limits.h>
//@ assigns \nothing;
int addition (int x, int y){
        return x + y;
}
//@ assigns \nothing;
int subtraction (int x, int y) {
    return x - y;
}
//@ assigns \nothing;
int main()
{
        int x = 2147483646;
        int y = 3;
        int result = subtraction(x,y);
        result = addition(x,y);
        return 0;
}
```

```
example.c:4 eva    signed overflow. assert x + y ≤ 2147483647;
```

Figure 3.3 Example of a graphical interface: FRAMA-C-GUI.

From the normalized source code view in Figure 3.3, we can see the result of the attempted verification. A couple of properties were successfully verified, one was verified partially and one was deemed invalid. The red parts of the normalized code highlight lines which will never be executed or weren't reached by the framework. In this case, they are red because `EVA` stopped the process upon evaluating the assertion in the ADDITION function.

Each property, that the framework attempted to prove, is marked with a symbol of a colored circle.

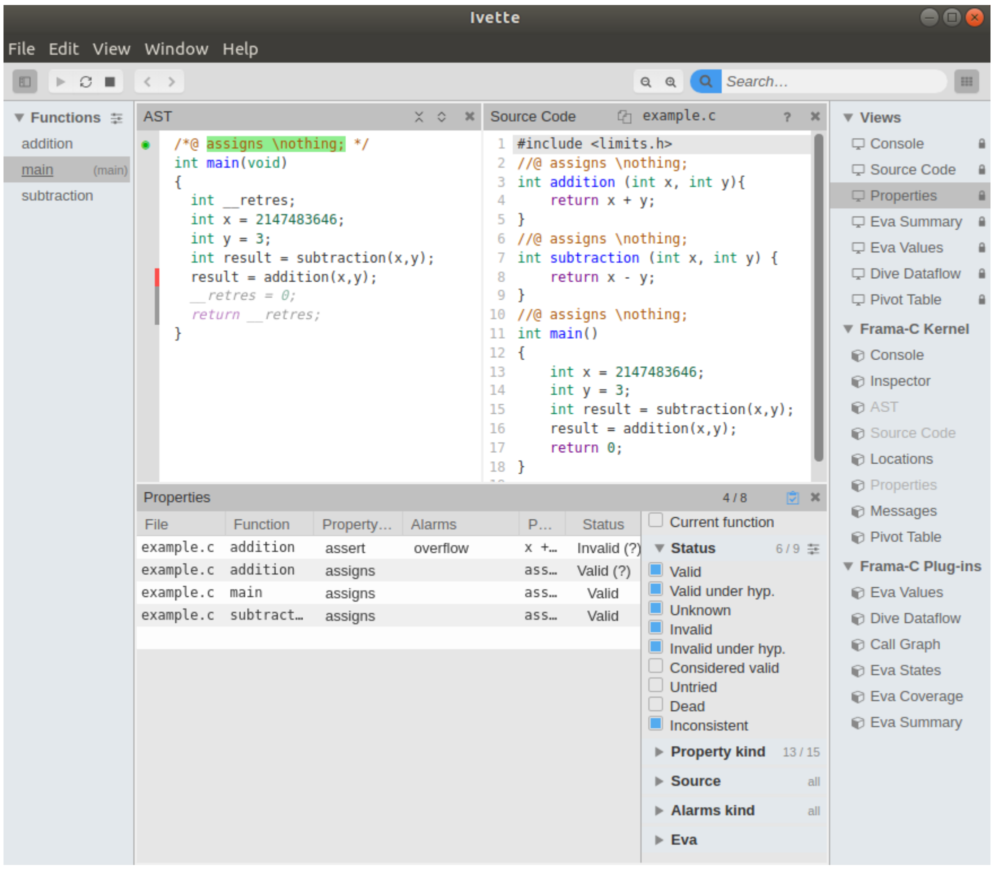
- A green circle means the property is always valid.
- Half green, half orange circle means the property is valid under hypotheses. This means it is verified, but has dependencies with unknown status.
- Half green, half black circle denotes that the verification process never reached the property because that part of the code will never execute.
- A yellow circle symbolizes a failure in the verification of the property.
- Half red, half yellow circle stands for an invalid property.
- An empty blue circle means verification was not attempted.
- A half green, half blue circle means `Frama-c` considers the property valid but did not attempt verification; leaving the proof to be done outside the framework. This can be seen with axiomatic properties.

Viewing results using `Ivette`

`Ivette` is a new Graphical User Interface developed for `Frama-c`, meant to replace the old GTK-based `frama-c-gui` user interface. The GUI desktop application uses HTML5 and NodeJS [23] JavaScript runtime engine of the Electron [24] platform. The entire GUI code base is written in TypeScript [25].

The first preview came out with `Frama-c` version 25.0 (Manganese). Its current interface, as seen in Figure 3.3, is based on modifiable **views** comprised of components. This design offers much greater versatility compared to the old `frama-c-gui` interface.

However, `Ivette` is still several versions away from becoming the default `Frama-c` GUI, as it currently supports only the `EVA` plug-in. Other plug-ins, including `WP`, will follow in future versions.



```
/*@ assigns \nothing; */
int main(void)
{
  int __retres;
  int x = 2147483646;
  int y = 3;
  int result = subtraction(x,y);
  result = addition(x,y);
  __retres = 0;
  return __retres;
}
```

```
#include <limits.h>
//@ assigns \nothing;
int addition (int x, int y){
    return x + y;
}
//@ assigns \nothing;
int subtraction (int x, int y) {
    return x - y;
}
//@ assigns \nothing;
int main()
{
    int x = 2147483646;
    int y = 3;
    int result = subtraction(x,y);
    result = addition(x,y);
    return 0;
}
```

File	Function	Property...	Alarms	P...	Status
example.c	addition	assert	overflow	x +...	Invalid (?)
example.c	addition	assigns		ass...	Valid (?)
example.c	main	assigns		ass...	Valid
example.c	subtract...	assigns		ass...	Valid

Figure 3.4 Example of a graphical interface: `Ivette`.

Chapter 4

Hoare Logic

Hoare logic, or Floyd–Hoare logic, is a formal system for reasoning about properties of programs [26]. The original ideas were proposed by Robert W. Floyd in 1967, who had developed a similar system for flowcharts [27]. The theory and examples in this chapter are adapted from Hoare's paper *An axiomatic basis for computer programming*[26] and the text *ACSL by Example* [28].

In his original paper, Hoare introduced a new notation:

$$P\ \{Q\}\ R \tag{4.1}$$

This notation has evolved over the years to:

$$\{P\}\ Q\ \{R\} \tag{4.2}$$

Where P is a logical expression called a *precondition*, Q is a program, and R is a logical expression called the *postcondition*, which describes the result of the execution of Q.

The triple (4.2), known today as the **Hoare Triple**, has become the basis of Hoare logic. It describes how the execution of code changes the state of the computation.

The precondition P represents the requirements needed for the proper execution of code Q, and it is up to the caller to guarantee that the precondition holds. The entire triple can be understood as "If the assertion P is true before initiation of code Q, then the assertion R will be true upon its completion."

An example of the Hoare triple, written in C, can be seen in code Code listing 4.1. There, the precondition is that the number x is odd, the code increases the value by one, and the postcondition states that x is now even.

Code listing 4.1 Hoare triple in C - Example 1

```
//@ assert x % 2 == 1;
x++;
//@ assert x % 2 == 0;
```

Notice that the code does not perform any form of validation of whether x is odd, as the precondition states. This property of x has to be ensured before this code is executed.

Now consider the Code listings 4.2. In this example, the code is the same as in Code listing 4.1, but both the precondition and the postcondition differ. There is no single correct precondition or postcondition for code Q. The choice of the formulas depends on the property that is to be verified.

Also, notice that the variable y does not appear in the code of the second example. The pre- and postconditions can be relative to the state of other variables.

Code listing 4.2 Hoare triple in C - Example 2

```
//@ assert 0 <= x <= y;
x++;
//@ assert 0 <= x <= y + 1;
```

The assignment rule

The assignment rule, in the form

$$P\{x \rightarrow e\} \tag{4.3}$$

substitutes each occurrence of variable x in the predicate P by the expression e. For example, given the following predicate P

$$P = \{x > 0 \wedge x < z\} \tag{4.4}$$

and the rule $P\ \{x \rightarrow y{+}1\}$, the resulting substitution would be

$$P = \{y + 1 > 0 \wedge y + 1 < z\} \tag{4.5}$$

An example of this rule used in C is in the Code listing 4.3.

Code listing 4.3 Example of the assignment rule in C

```
//@ assert y+1 > 0 && y+1 < z;
x = y+1;
//@ assert x > 0 && x < z;
```

The sequence rule

The sequence rule, shown in Code listing 4.4, combines two codes Q and S to a single piece of code Q ; S. This transformation is only possible when the postcondition of Q is identical to the precondition of S.

Code listing 4.4 The sequence rule

```
//@ assert P;           //@ assert R;          //@ assert P;
Q;              and     S;              ⟶     Q ; S;
//@ assert R;           //@ assert T;          //@ assert T;
```

The implication rule

The implication rule, shown in Code listing 4.5, allows to tighten the precondition P and weaken the postcondition R. This is possible if $R \implies R'$ and $P' \implies P$.

■ **Code listing 4.5** The implication rule

```
//@ assert P;          //@ assert P';
Q;                 ⟶  Q;
//@ assert R;          //@ assert R';
```

The choice rule

The choice rule allows the verification of conditional statements in the form

```
if (C) X;
else Y;
```

The rule unites Hoare triples with identical postconditions, but differing codes and preconditions (shown in the Code listing 4.6), into a single Hoare triple seen in Code listing 4.7.

■ **Code listing 4.6** Hoere triples used in the choice rule

```
//@ assert P && C;            //@ assert P && !C;
Q;                   and       S;
//@ assert R;                  //@ assert R;
```

■ **Code listing 4.7** Result of choice rule

```
//@ assert P;
if (C)   X;
else     Y;
//@ assert R
```

The loop rule

The loop rule, shown in Code listing 4.8 serves to verify a **while** loop. It requires an appropriate formula P, called a *loop invariant*, which is true at every iteration of the loop.

This rule does not guarantee that the loop will ever terminate, it merely states that if it does, the postcondition will hold.

■ **Code listing 4.8** The loop rule

```
//@ assert P;
while (B)  \{
Q;
\}
//@ assert P;
```

Derived rules

The mentioned rules do not cover many statements allowed in C. However, such statements can be rewritten into a form that is semantically equivalent to those covered by Hoare rules.

A **switch** statement can be rewritten as a sequence of nested **if** statements, which are covered by the choice rule.

Of special note is the expression of a **for** loop

```
for (P; Q; R) {
    S;
}
```

as a **while** loop

```
P;
while (Q) {
    S;
    R;
}
```

which will be heavily used in Chapter 6.

The only statement that cannot be rewritten this way is **goto**, which can render an entire source code unverifiable using Hoare logic. Programs with arbitrary jumps can, however, be analysed using the calculus proposed by Robert W. Floyd [27].

Chapter 5

ANSI/ISO C Specification Language

In this chapter, we will go over the ANSI/ISO C Specification Language (ACSL) [29] and explain what it is, how it relates to `Frama-c`, and how to use it in the process of formal verification.

The main resource of this chapter is the documentation *ANSI/ISO C Specification Language* [30].

ACSL is a *Behavioral Interface Specification Language* [31] implemented in `Frama-c`. It specifies the behavioural properties of C programs in terms of preconditions, postconditions, and invariants. This is achieved through the use of annotations in the source code comments. These annotations are then used by `Frama-c` to verify that the code adheres to the specifications.

Lexical rules

To signal to `Frama-c` that a comment is an ASCL annotation, the symbol '@' is used at the beginning of the comment, as can be seen in Code listing 5.1. At any other place, the symbol '@' is equivalent to a space character.

Comments can be put in the ACSL annotations, and they follow the single-line C++ comment format.

Code listing 5.1 Forms of single-line and multi-line annotation comments

```
//@ A single-line annotation comment

/*@
    A multi-line annotation comment
*/
```

5.1 Function contract

A fundamental concept of ACSL is the *function contract*, which represents the Hoare triple. The preconditions P and postconditions R are specified in the annotations, while the code Q is a specific function f. This concept forms a contract between the caller and the function f. Each caller of f must uphold the preconditions, while the f guarantees that the postconditions are true on return.

The clause REQUIRES signifies that the following predicate P is a precondition. The clause ENSURES combined with a predicate R represents a postcondition. We will now take a look at a specific example of a simple function contract at Code listing 5.2.

Code listing 5.2 Example of a simple function contract

```
/*@
    requires y != 0;
    ensures \result == x / y;
*/
int division(int x, int y) { return x / y; }
```

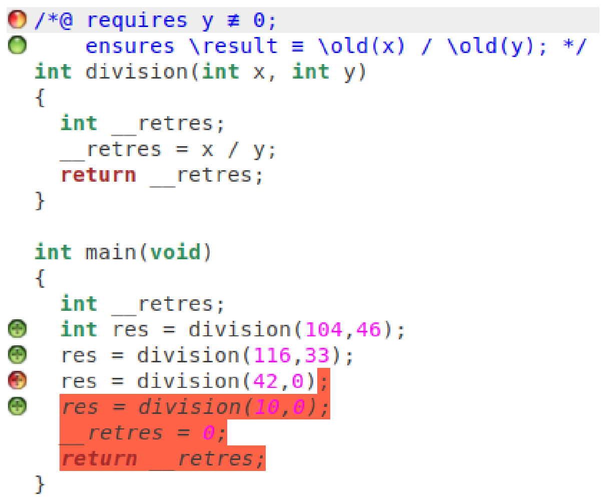
The precondition in the example states that y cannot be equal to 0. Again, the function does not check whether or not the precondition holds; if it gets called with $y = 0$, it will proceed and cause a division by zero error. It is up to the caller of the function to guarantee that y will not be 0.

The ENSURE clause uses the construct \RESULT which refers to the return value of the function. This construct can only be used in contracts of functions that do not return VOID. In this case, it states that DIVISION returns the expected value x/y.

Now we will expand the program by adding a MAIN function, which will call the DIVISION function, and we will attempt a verification. See Code listing 5.3 for the expanded source code.

Code listing 5.3 Example source code

```c
#include <limits.h>
#include <stdio.h>

/*@
    requires y != 0;
    ensures \result == x / y;
*/
int division(int x, int y) { return x / y; }

int main()
{
    int res = division(104, 46);
    res = division(116, 33);
    res = division(42, 0);
    res = division(10, 0);
    return 0;
}
```



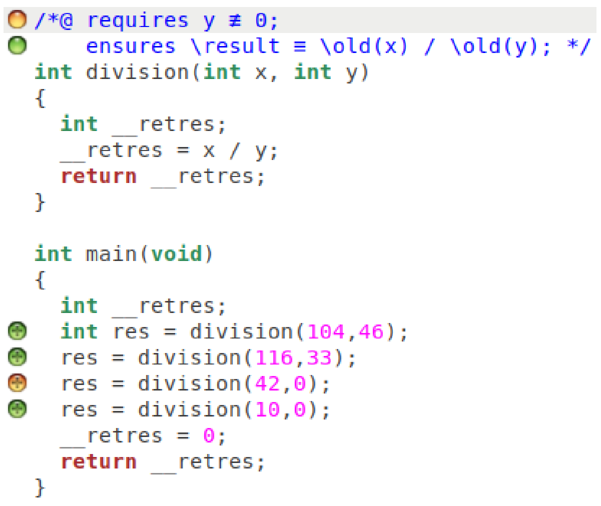
```c
/*@ requires y ≢ 0;
    ensures \result ≡ \old(x) / \old(y); */
int division(int x, int y)
{
  int __retres;
  __retres = x / y;
  return __retres;
}

int main(void)
{
  int __retres;
  int res = division(104,46);
  res = division(116,33);
  res = division(42,0);
  res = division(10,0);
  __retres = 0;
  return __retres;
}
```

Figure 5.1 Output of attempted verification in FRAMA-C-GUI using WP and EVA plug-ins.

Figure 5.1 displays the normalized source code view. Frama-c has determined that the precondition for the division function is valid in three out of four calls. The framework correctly identified a violation of the precondition in the third call.

Because Frama-c used the EVA plug-in, the analysis concluded after the third call. The fourth call, which was not analyzed, was still marked as valid by Frama-c.

If the EVA plug-in is not used, the error is much less apparent, as can be seen in Figure 5.2.



```
/*@ requires y ≢ 0;
    ensures \result ≡ \old(x) / \old(y); */
int division(int x, int y)
{
  int __retres;
  __retres = x / y;
  return __retres;
}

int main(void)
{
  int __retres;
  int res = division(104,46);
  res = division(116,33);
  res = division(42,0);
  res = division(10,0);
  __retres = 0;
  return __retres;
}
```

Figure 5.2 Output of attempted verification in FRAMA-C-GUI using the WP plug-in.

In this case, `Frama-c` has evaluated the precondition for the third call as 'unknown' and continued. This led to a contradiction, and according to the principle of explosion, "from contradiction, anything follows."

Another construction prevalent in function contracts is \OLD(e), which denotes the value of predicate or term e in the pre-state. `Frama-c` automatically adds this construction to the normalized code, as can be seen in Figure 5.2. Note that working with the values of both x and y in their pre-state makes sense, as they are passed by value. Any modification made to them would not propagate to the rest of the program. The most common use of the \OLD(e) construction is when e is a pointer. In such cases, the ENSURE clauses can include both pre-state a post-state values of the same variable. An example of the \OLD(e) construction paired with a pointer can be seen in Code listing 5.4.

Code listing 5.4 Example of the \OLD(e) construction.

```
/*@
    requires \valid(x) && *x < INT_MAX;
    ensures *x == \old(*x) + 1;
*/
void increment(int* x) { ++(*x); }
```

The normalized code of line 3 is as follows

```
//@ ensures *\old(x) ≡ \old(*x) + 1;
```

and states that the post-state value of x is equal to the pre-state value of x plus one.

Pointers and Arrays

Pointers are an integral part of the C language; therefore, `Frama-c` is well-equipped to handle them. Along with \OLD(e) construct, the framework also provides two built-in predicates to address the validity of pointers and arrays. A pointer p is *valid* if dereferencing p produces a definite value according to the C standard. An array a of size n is valid if every pointer from a+0 to a+n-1 is valid.

- \VALID(s), where s is a set of l-values. This predicate holds if and only if dereferencing any pointer $p \in s$ is safe, both for reading from $*p$ and writing to it.
- \VALID_READ(s), where s is a set of l-values. It holds if and only if it is safe to read from all pointers p in the s.

See Code listing 5.5 for an example showcasing the use of these constructions.

Code listing 5.5 Example of \valid and \valid_read

```
/*@
    requires \valid(p);
    requires \valid(x + (0 .. x_size-1))
    requires \valid_read(y + (0 .. y_size-1));
    requires x_size > 0 && y_size > 0;
*/
void foo(   int* x, int x_size,
            int* y, int y_size, int* p);
```

Behaviors

Behaviors expand function contracts by specifying what behavior can be expected from a function based on parameters. Each behavior can have additional ENSURE clauses and the selection of which behavior applies to a specific call is done through the ASSUMES clause. Behaviors can be labelled with:

- COMPLETE BEHAVIORS to indicate that the behaviors cover all contexts,
- DISJOINT BEHAVIORS to indicate that the behaviors cover disjoint cases.

`Frama-c` will attempt to verify both labels. See Code listing 5.6 for an example of behavior usage.

Code listing 5.6 Example of the bahviors

```
/*@
    requires \valid(x) && \valid(a) && \valid(b);
    behavior a_less:
        assumes *a < *b;
        ensures *x == *a;
    behavior b_less_eq:
        assumes *a >= *b;
        ensures *x == *b;
    complete behaviors a_less, b_less_eq;
    disjoint behaviors a_less, b_less_eq;
*/
void set_pointer_to_min(int* x, int* a, int* b){
    if(*a < *b)
        *x = *a;
    else
        *x = *b;
}
```

Assign clause

Every example of ACSL annotations shown up to this point was missing an important part. The ASSIGNS clause. This clause states if and how a function modifies memory outside its local variables. If the clause is missing, the function can modify any visible variable, which can significantly complicate the verification process. If no such memory is altered then the \NOTHING clause can be used to indicate this. See Code listing 5.7 for an example of the ASSIGNS clause.

An experimental feature of ACSL expands the ASSIGNS clause with an additional clause: \FROM. This clause indicates that assigned values can only depend on the locations specified with the \FROM clause.

Code listing 5.7 Example of the ASSIGNS clause

```
/*@
    requires \valid(p);
    requires \valid(x + (0 .. x_size-1));
    requires x_size > 0;
    assigns x[0 .. x_size-1], *p;
*/
void foo(int* x, int x_size, int* p);
```

5.2 Loops

Loops pose a challenge for verification and analysis, as the number of iterations of any loop may be unknown. Therefore, ACSL has a clause LOOP INVARIANT *predicate P* which solves the problem by specifying the correct state of the program in each iteration. The predicate P in this clause must hold before entering the loop and at the start of every iteration.

A second important clause is the LOOP ASSIGNS *locations L* clause which lists parts of the memory the loop manipulates.

The final loop clause used in this thesis is LOOP VARIANT *integer m*. This is an optional clause with the following semantics: the value of m at the beginning of each iteration must be nonnegative, and for each loop iteration that ends normally or with a *continue* statement, the value of m at the end of the iteration must be smaller than its value at the beginning of the iteration.

The code in Code listing 5.8 shows the annotations for a **for** loop and a nested **for** loop, each using all the loop clauses mentioned.

Code listing 5.8 Example of the LOOP clauses

```
/*@
    loop invariant 0 <= i <= NumOfVer;
    loop invariant \forall int k,l; 0 <= k < i
            ==> 0 <= l < NumOfVer ==> graph[k][l] == 0;
    loop assigns i,
            graph[0 .. NumOfVer - 1][0 .. NumOfVer - 1];
    loop variant NumOfVer-i;
*/
for (int i = 0; i < NumOfVer; i++){
    /*@
        loop invariant 0 <= j <= NumOfVer;
        loop invariant \forall int m,n; 0 <= m < i
                ==> 0 <= n < j ==> graph[m][n] == 0;
        loop assigns j,
                graph[0 .. NumOfVer - 1][0 .. NumOfVer - 1];
        loop variant NumOfVer-j;
    */
    for (int j = 0; j < NumOfVer; j++)
        graph[i][j] = 0;
}
```

5.3 Predicates

ACSL allows for the declaration of new logic predicates. These allow for the expression of complex conditions through a simple logical expression. This improves the readability, modularity and maintainability of formal specifications.

Code listing 5.9 shows the definition of a new predicate which specifies that all elements in a 2D array are within a specified range.

Code listing 5.9 Example of predicate definition

```
/*@
    predicate
    Valid_2D_Array_Values{L}(   int* array,
                                int size_of_array,
                                int lower_bound
                                int upper_bound) =
        \forall integer i, j;   0 <= i < size_of_array
                             && 0 <= j < size_of_array ==>
            (*( graph + i * size_of_array + j) >= lower_bound
        && *(   graph + i * size_of_array + j) <= upper_bound);
*/
```

A predicates may also be defined inductively in the following form

```
/*@ inductive P(x_1,...,x_n) {
     case c_1 : p_1;
...
     case c_k : p_k;
}
*/
```

where each c_i is an identifier and each p_i is a proposition.

5.4 Labels and Construct \at

Statements will oftentimes require an additional construct \AT(e, *id*) referring to the value of the expression e in the state at label *id*.

ACSL comes with seven predefined logic labels, the three used in this thesis are: *Pre*, *LoopEntry*, *LoopCurrent*.

- The *Pre* label can be used in statement annotations and refers to the pre-state of the enclosing function.
- The *LoopEntry* and *LoopCurrent* labels can be used in loop annotations and loop statements. *LoopEntry* refers to the state prior to first loop entry, while *LoopCurrent* refers to the state at the beginning of the current loop iteration.

Chapter 6

Verified implementation of Dijkstra's algorithm

This chapter will show and discuss a specific implementation of Dijkstra's algorithm from Chapter 2, using the programming language C. Then, we will add ACSL annotations and attempt to verify the implementation with the `Frama-c` platform.

The first goal of the verification is to show a complete absence of runtime errors and undefined behaviors. Later, we will attempt to prove the correctness of the results and showcase the problems that occurred in the process.

6.1 Implementation

The entire program was written using static memory. Though it is a suboptimal approach in terms of efficiency, it allows Frama-c to perform a more comprehensive analysis. As of writing this thesis, verification of dynamic memory is an experimental feature not fully implemented in Frama-c.

The complete source code is split into three files. File *main.c* contains the initial entry point and is responsible for reading and storing user input, calling the `dijkstra()` function, and displaying the results.

The implementation of Dijkstra's algorithm itself is split into the two remaining files. The header file *dijkstra.h* contains the necessary constants, the data structure for representing nodes and the function prototypes. The source file *dijkstra.c* contains the implementation of each function declared in *dijkstra.h*.

6.1.1 main.c

The main purpose of the functions in *main.c* is to create and store a representation of a weighted graph. The edges of the graph are stored as an adjacency matrix in a 2D static array `static_graph`. The dimensions of the array are given by a constant `NumOfVer` declared in *dijkstra.h*. The nodes of the graph are stored in a static array, called `node_array`, of length `NumOfVer`.

The code of file *main.c* contains three function:

- `read_input()` which assigns an integer to `source_node` representing the index of the source node. Then it loads a set of integers representing the weights of the edges and assigns them to the 2D array `static_graph`. The allowed range for `source_node` is $0 \leq \texttt{source\_node} < \texttt{NumOfVer} - 1$. The weights of the graph are similarly constrained to a range from 0 to `MaxWeight`, a constant from *dijkstra.h*. When the weight of the edge is set to 0, it is understood as an absence of an edge. Before the function returns, it also eliminates any self-loop that may have been inserted. It does so by setting the entire diagonal of the matrix to zero.

- `print_solution()` is a function that displays the shortest-path estimates of each node in the graph. If the estimate is still at ∞, represented as INT_MAX, it states that the node is unreachable.

- `main()` is the entry point of the program. It declares the array `node_array` and the 2D array `static_graph`, then it calls the `dijkstra()` function and `print_solution()` function before exiting.

6.1.2 dijkstra.h

Contents of *dijkstra.h* can be seen in Code listing 6.1. The file contains the definition of `struct node`, which holds two integers and a pointer:

- `int distance` is the shortest-path estimate, used as described in Section 2.1.

- `int id` is a unique integer assigned to every node. It can be used as an index to access either the node from `node_array` or the corresponding row/column from `static_graph`.

- `struct node* parent` is a pointer to the node's predecessor.

Next, the file contains two constants, `NumOfVer` specifies the expected number of nodes in the given graph, and `MaxWeight` states the maximum allowed weight of any given edge.

Lastly, the file contains declarations of function prototypes, which are implemented in *dijkstra.c*. The function prototypes are as follows:

- `void initialize()` initializes the nodes in the array `node_array` with appropriate values.

- `int min_node()` iterates over an array of node pointers in the range from 0 to `int size` $\leq$ `NumOfVer`, and returns the index of the pointer to the node with the lowest `distance`.

- `void swap()` swaps the value of two pointers to `struct node`.

- `struct node* extract_min()` extracts the node with the lowest shortest-path estimate from the array `set_Q`, in the range from 0 to `q_size`. It then fixes the structure of the array and returns a pointer to the extracted node.

- `void relax()` is a variation on the RELAX procedure from Section 2.1. This function always replaces the shortest-path estimate of node v with the estimate of u plus the weight of the edge and sets u as the predecessor of v.

- `void dijkstra()` is the central function of the implementation. It calculates the shortest-path estimate of each node from the source node and stores their predecessors. The function does not return anything, but all the necessary information is stored in the array `node_array`.

Code listing 6.1 Contents of file *dijkstra.h*

```c
#ifndef DIJKSTRA_H
#define DIJKSTRA_H

#include <limits.h>
#define NumOfVer 9
#define MaxWeight 100

struct node {
    int distance;
    int id;
    struct node* parent;
};

void initialize(struct node node_array[NumOfVer]);

int min_node(struct node* set_Q[NumOfVer], int size);

void swap(struct node** pointer_1, struct node** pointer_2);

struct node* extract_min( struct node* set_Q[NumOfVer],
                          int q_size);

void relax(struct node* u, struct node* v, int weight);

void dijkstra(  struct node node_array[NumOfVer],
                int static_graph[NumOfVer][NumOfVer],
                int sourceNode);

#endif // DIJKSTRA_H
```

6.1.3 dijkstra.c

This file contains the definition of the function prototypes declared in *dijkstra.h*. The functions `swap`, `min_node`, and `extract_min` can be thought of as ”outside” the Dijkstra’s algorithm, as they are a consequence of the data structures used.

The `min_node` performs a linear search over an array of node pointers. The `swap` function swaps the values using a temporary variable.

The function `extract_min` relies on both the `min_node` and the `swap` functions. Because the min-priority queue `set_Q` is implemented as a static array, extracting an element does not decrease its size. Therefore, an integer `q_size` is maintained to track the number of elements considered active in the queue, starting from index 0. To remove an element, it is swapped with the element at index `q_size`−1. After the swap, `q_size` is decremented by one to indicate the removal of the element.

The source code of these three functions is in Code listing 6.2.

Code listing 6.2 `swap`, `min_node`, and `extract_min` functions from *dijkstra.c*

```c
#include <stdlib.h>
#include "dijkstra.h"

int min_node(struct node* set_Q[NumOfVer], int size){
    int min = 0;
    for (int i = 0; i < size; i++)
        if(set_Q[i] != NULL && set_Q[min] != NULL)
            if (set_Q[i]->distance < set_Q[min]->distance)
                min = i;
    return min;
}

void swap(struct node** pointer_1, struct node** pointer_2){
    struct node* tmp = *pointer_1;
    *pointer_1 = *pointer_2;
    *pointer_2 = tmp;
}

struct node* extract_min(struct node* set_Q[NumOfVer], int q_size){
    int index = min_node(set_Q, q_size);
    struct node* res = set_Q[index];
    swap(&set_Q[index], &set_Q[q_size-1]);
    return res;
}
```

The remaining three functions: `dijkstra, relax`, and `initialize` closely resemble the procedures introduced in Chapter 2. The differences are:

- The IF statement from the RELAX procedure was moved to the `dijkstra` function.
- The INITIALIZE-SINGLE-SOURCE sets the shortest-path estimate of source node s to zero. This functionality was moved from `initialize` to the `dijkstra` function.
- Unlike the DIJKSTRA procedure, the `dijkstra` function does not maintain the set S. This set is useful for proving the correctness of the algorithm but otherwise serves no purpose. However, the function uses the size that set S should have at any given point as a control variable of the main **for** loop.
- The lack of adjacency lists forces the `dijkstra` function to iterate over every node v and check whether there is an edge between v and the current node u.
- The DECREASE-KEY operation is omitted, as the elements in the array `set_Q` are not keyed by any value.

See Code listing 6.3 for the source code of these functions.

Code listing 6.3 `initialize`, `relax`, and `dijkstra` functions from *dijkstra.c*

```c
void initialize(struct node node_array[NumOfVer]) {
    for (int i = 0; i < NumOfVer; i++){
        node_array[i].distance = INT_MAX;
        node_array[i].parent = NULL;
        node_array[i].id = i;
    }
}

void relax(struct node* u, struct node* v, int weight){
    v->distance = u->distance + weight;
    v->parent = u;
}

void dijkstra(  struct node node_array[NumOfVer],
                int static_graph[NumOfVer][NumOfVer],
                int sourceNode){

    struct node* set_Q[NumOfVer] = { NULL };
    int q_size = NumOfVer;
    node_array[sourceNode].distance = 0;
    node_array[sourceNode].parent = &node_array[sourceNode];

    for (int i = 0; i < NumOfVer; i++)
        set_Q[i] = &node_array[i];
    struct node* u;

    for (int s_size = 0; s_size < NumOfVer; ++s_size){
        u = extract_min(set_Q, q_size);
        q_size--;
        for (int i = 0; i < NumOfVer; i++)
            if( static_graph[u->id][i] > 0 && u->distance
                < node_array[i].distance - static_graph[u->id][i])
                relax(u, &node_array[i], static_graph[u->id][i]);
    }
}
```

6.2 Annotation with ACSL

With the implementation ready, we will now specify the desired behavior using ACSL annotations, focusing on a lack of runtime errors and proper handling of arrays and pointers. We will start by crafting the function contracts.

6.2.1 Function contracts

Contract: `swap()`

The function has two pointers as parameters and the only requirement of the function is their validity. The only location in memory the function manipulates is the location of the two pointers, and lastly, the function guarantees a swap. The complete contract is at Code listing 6.4.

Code listing 6.4 Function contract of `swap`

```
/*@
    requires valid:      \valid(pointer_1)
                    &&   \valid(pointer_2);

    assigns *pointer_1, *pointer_2;

    ensures swap:        *pointer_1 == \old(*pointer_2)
                    &&   *pointer_2 == \old(*pointer_1);
*/
void swap(struct node** pointer_1, struct node** pointer_2);
```

Contract: `relax()`

The next function is `relax`. The preconditions will be the validity of pointers u and v, the appropriate value of $weight$, and assurance that the shortest-path estimate of u + $weight$ will not cause an overflow by exceeding INT_MAX. Next, the function modifies v, and lastly, it guarantees that u will become the predecessor of v and v's shortest-path estimate is decreased. The contract is at Code listing 6.5

Code listing 6.5 Function contract of `relax`

```
/*@
    requires valid:      \valid(u) && \valid(v);
    requires overflow:   u->distance < v->distance - weight;
    requires bound:      0 < weight < INT_MAX;

    assigns *v;

    ensures parent:      v->parent == u;
    ensures decrease:    v->distance < \old(v->distance);
*/
void relax(struct node* u, struct node* v, int weight);
```

Contract: initialize()

The `initialize` function requires a valid array of nodes, which it also modifies. The function guarantees that the attributes of each node get set to proper default values.

Code listing 6.6 Function contract of `initialize`

```
/*@
    requires valid_array:   \valid(node_array + (0 .. NumOfVer-1));

    assigns node_array[0 .. NumOfVer-1];

    ensures valid_distance: \forall int i; 0 <= i < NumOfVer
                              ==> node_array[i].distance == INT_MAX;
    ensures valid_id:       \forall int i; 0 <= i < NumOfVer
                              ==> node_array[i].id == i;
    ensures valid_parent:   \forall int i; 0 <= i < NumOfVer
                              ==> node_array[i].parent == NULL;
*/
void initialize(struct node node_array[NumOfVer]);
```

Contract: `min_node()`
The preconditions for this function are the validity of the array *set_Q* from 0 to *size*, where *size* is greater than 0 and less or equal to `NumOfVer`. The function is non-mutating, so it does not manipulate memory outside its stack.

The first postcondition is simple; it states that the return value is from the interval $\langle 0, \text{size})$. However, the other postconditions are more challenging to specify. They need to express that the node at index *\result* has the lowest shortest-path estimate among all nodes in the array. Additionally, if multiple nodes share the same shortest-path estimate, the function returns the index of the first such node.

To properly express these postconditions, we will introduce three predicates: `Lower_Bound`, `Strict_Lower_Bound`, and `Min_Element`.

Code listing 6.7 Array predicates

```
/*@
    predicate
    Lower_Bound{L}(          struct node** nodes,
                             integer n, struct node* value)
        = \forall integer i; 0 <= i < n ==>
            value->distance <= nodes[i]->distance;

    predicate
    Strict_Lower_Bound{L}(   struct node** nodes,
                             integer n, struct node* value)
        = \forall integer i; 0 <= i < n ==>
            value->distance < nodes[i]->distance;

    predicate
    Min_Element{L}(struct node** nodes, integer n, integer min)
        = 0 <= min < n && Lower_Bound(nodes, n, nodes[min]);
*/
```

Using these predicates, we can complete the function contract as seen in Code listing 6.8.

Code listing 6.8 Function contract of `min_node`

```
/*@
    requires bound:         0 < size <= NumOfVer;
    requires valid_array:   \valid_read(set_Q + (0..size-1));

    assigns \nothing;

    ensures valid_result:   0 <= \result < size;
    ensures minimum:        Min_Element(set_Q, size, \result);
    ensures first:          Strict_Lower_Bound(set_Q,
                                     \result, set_Q[\result]);
*/
int min_node(struct node* set_Q[NumOfVer], int size);
```

Contract: `extract_min()`

Now that both `swap` and `min_node` have their contracts, we can form the contract for `extract_min`.

The function requires a valid array *set_Q* and proper an integer *size* in a proper range. The function then manipulates *set_Q* via the `swap` function. Lastly, it ensures *set_Q* is still valid and that the returned pointer points to a node with the lowest shortest-path estimate.

Code listing 6.9 Function contract of `extract_min`

```
/*@
    requires valid_array:   \valid(set_Q + (0 .. NumOfVer-1));
    requires bound: 0 < q_size <= NumOfVer;

    assigns set_Q[0 .. NumOfVer-1];

    ensures Lower_Bound(set_Q, q_size, \result);
    ensures \valid(set_Q + (0 .. NumOfVer-1));
*/
struct node* extract_min(struct node* set_Q[NumOfVer], int q_size);
```

Contract: `dijkstra()`

To avoid runtime errors, the `dijkstra` function requires a valid array *node_array*, a valid 2D array *graph*, and an integer *sourceNode* from a proper interval. The function modifies only the array *node_array* and ensures it stays valid.

Code listing 6.10 Function contract of `dijkstra`

```
/*@
    requires valid_array: \valid(node_array + (0 .. NumOfVer-1));
    requires valid_graph: \valid_read(graph + (0 .. NumOfVer-1))
                          && \forall int i; 0 <= i < NumOfVer ==>
                          \valid_read(graph[i]+(0.. NumOfVer-1));
    requires valid_index: 0 <= sourceNode < NumOfVer;

    assigns node_array[0 .. NumOfVer-1];

    ensures valid_array: \valid(node_array + (0 .. NumOfVer-1));
*/
void dijkstra(  struct node node_array[NumOfVer],
                int graph[NumOfVer][NumOfVer], int sourceNode);
```

6.2.2 Loop annotations

With the function contracts ready, we now can delve into the code of each function and add loop annotations that will allow `Frama-c` to verify the ASSIGNS and ENSURES clauses.

The functions `swap`, `relax`, and `extract_min`, do not contain any cycles, so they are ready for verification. Out of the remaining three, we will start with `initialize`.

Loop annotations: `initialize()`

The function contains a single **for** loop which makes three assignments to a node from *node_array* during every iteration.

The invariant of the loop states that every node processed by the loop has its attributes—*distance*, *parent*, and *id*—set to INT_MAX, NULL, and *i*, respectively.

Code listing 6.11 Loop annotations of `initialize`

```
void initialize(struct node node_array[NumOfVer])
{
    /*@
      loop invariant bound:      0 <= i <= NumOfVer;
      loop invariant distance:  \forall int k; 0 <= k < i
                                    ==> node_array[k].distance
                                        == INT_MAX;
      loop invariant parent:    \forall int k; 0 <= k < i
                                    ==> node_array[k].parent
                                        == NULL;
      loop invariant id:        \forall int k; 0 <= k < i
                                    ==> node_array[k].id == k;

      loop assigns i, node_array[0 .. NumOfVer-1];

      loop variant end: NumOfVer-i;
    */
    for (int i = 0; i < NumOfVer; i++)
    {
        node_array[i].distance = INT_MAX;
        node_array[i].parent = NULL;
        node_array[i].id = i;
    }
}
```

Loop annotations: `min_node()`
The function contains a single **for** loop which assigns only local variables. The loop invariant states that the node at $set\_Q[min]$ has the lowest shortest-path estimates of all scanned nodes and is the first node to have such a value.

The annotations of the invariant once again use the axiomatic predicates defined in Code listing 6.7.

Code listing 6.12 Loop annotations of `min_node`

```
int min_node(struct node* set_Q[NumOfVer], int size)
{
    int min = 0;
    /*@
        loop invariant bound:       0 <= i <= size;
        loop invariant min:         0 <= min < size;
        loop invariant valid_array: \forall int k; 0 <= k < i
                                        ==> \valid(set_Q[k]);
        loop invariant lower:       Lower_Bound(set_Q,
                                        i, set_Q[min]);
        loop invariant first:       Strict_Lower_Bound(set_Q,
                                        min, set_Q[min]);
        loop assigns min, i;
        loop variant size-i;
    */
    for (int i = 0; i < size; i++) {
        if(set_Q[i] != NULL && set_Q[min] != NULL){
            if (set_Q[i]->distance < set_Q[min]->distance)
                min = i;
        }
    }
    return min;
}
```

Loop annotations: `dijkstra()`

The `dijkstra` function contains three **for** loops, first at lines 23–24, second at lines 27–34, and the third is nested at lines 30–33, in Code listing 6.3.

The first loop fills the min-priority queue *set_Q* with nodes from *node_array*. The second loop extracts node *u* from *set_Q* and, using the third loop, it relaxes each outgoing edge from node *u*.

The annotations for the first loop are straightforward.

Code listing 6.13 Annotations of the first loop in `dijkstra`

```
/*@
    loop invariant bound:       0 <= i <= NumOfVer;
    loop invariant assigned:    \forall int k; 0 <= k < i
                                    ==> set_Q[k] == &node_array[k];
    loop assigns i, set_Q[0 .. NumOfVer-1];
    loop variant NumOfVer - i;
*/
for (int i = 0; i < NumOfVer; i++)
    set_Q[i] = &node_array[i];
```

Before annotating the second loop at lines 27–34, we analyse and annotate the nested loop at lines 27–34. This loop calls the `relax` function, which according to its contract, modifies the second argument. Therefore, the loop assigns *node_array*.

The loop at lines 27—34 maintains an invariant stating that the number of nodes in *set_Q* plus the number of nodes with their final shortest-path estimate determined equals the overall number of nodes in the graph. The loop assigns the current node *u* and *set_Q* through extraction on line 28, *q_size* on line 29 and *node_array* through the inner loop.

See Code listing 6.14 for annotations of the second and third loops.

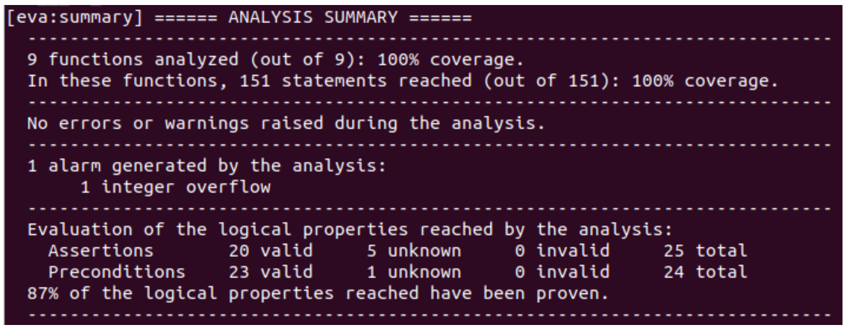
Code listing 6.14 Annotations of the second and third loops in `dijkstra`

```
/*@
loop invariant bound: 0 <= q_size <= NumOfVer;
loop invariant q_size + s_size == NumOfVer;
loop assigns    u, set_Q[0 .. NumOfVer-1],
                q_size, s_size, node_array[0 .. NumOfVer-1];

loop variant NumOfVer - s_size;
*/
for (int s_size = 0; s_size < NumOfVer; ++s_size){
    u = extract_min(set_Q, q_size);
    q_size--;
    /*@
        loop invariant bound:      0 <= i <= NumOfVer;
        loop assigns i, node_array[0 .. NumOfVer-1];
        loop variant end: NumOfVer-i;
    */
    for (int i = 0; i < NumOfVer; i++)
        if(static_graph[u->id][i] > 0 && u->distance
            < node_array[i].distance-static_graph[u->id][i])
            relax(u, &node_array[i], static_graph[u->id][i]);
}
```

6.3 First Verification

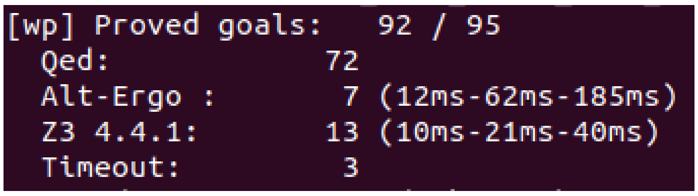
With the added annotations, the program is ready for verification. We will use the plug-ins `WP`, `RTE`, and `EVA`, combined with solvers `Alt-Ergo` and `Z3` introduced in Section 3.3.

The results of the analysis are seen in Figures 6.1 and 6.2.



```
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------------------
  9 functions analyzed (out of 9): 100% coverage.
  In these functions, 151 statements reached (out of 151): 100% coverage.
  ----------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------------------
  1 alarm generated by the analysis:
       1 integer overflow
  ----------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        20 valid      5 unknown      0 invalid      25 total
    Preconditions     23 valid      1 unknown      0 invalid      24 total
  87% of the logical properties reached have been proven.
  ----------------------------------------------------------------------------
```

Figure 6.1 EVA summary of the analysis. The analysis has reached all statements in the source code and generated one integer overflow warning. This warning is in the `relax` function, specifically at the assignment of the new shortest-path estimate. However, the annotation generated by this warning was verified, proving that an overflow cannot occur.



```
[wp] Proved goals:   92 / 95
  Qed:              72
  Alt-Ergo :         7 (12ms-62ms-185ms)
  Z3 4.4.1:         13 (10ms-21ms-40ms)
  Timeout:           3
```

Figure 6.2 WP summary of the analysis. All but three goals have been proven valid. The three goals that had reached the timeout limit were proof obligations in the `readu_input` function. These failures occurred due to the use of the unsafe function `scanf` but have overall no bearing on this implementation of Dijsktra's algorithm, and therefore, are not even listed in this thesis.

It is worth noting, that the required precision set by -EVA-PRECISION has to be adjusted based on the number of nodes specified by the `NumOfVer` in *dijkstra.h*. The more nodes we allow the graph to have, the higher the precision has to be to match the increase in space complexity. With the increase of both the number of nodes allowed and the precision of the analysis, the time it takes `Frama-c` to complete the analyses grows drastically. For example, with only twenty nodes and a precision of two, the process takes only a couple of seconds. Whereas with one hundred nodes and a precision of five, the analysis took several minutes. An increase to thousands of nodes and a precision of eight or more would lead to the analysis lasting hours.

6.4 Additional Annotations

In the first verification, we have shown that the implementation properly works with memory and we did not reveal any runtime errors. In the second verification, we will show the after running the `dijkstra` function, the shortest-path estimate of every node will either stay the same or decrease.

First, we will add the following ENSURES clause to the function contract of `dijkstra` and label this clause ESTIMATE_DECREASE

```
/*@
ensures estimate_decrease: \forall int i;
                           0 <= i < NumOfVer
                           ==> node_array[i].distance
                           <= \old(node_array[i].distance);
*/
```

Next, we move the call of the function `initialize` outside `dijkstra`, otherwise, the construct \OLD in the newly added ENSURES clause would reference uninitialized values. Furthermore, this change requires an addition of new REQUIRES clauses to the contract. These will specify that the attributes of all nodes were initialized properly. See Code listing 6.15 for the full function contract.

Code listing 6.15 The complete function contract of the *dijkstra* function

```
/*@
    requires valid_array: \valid(node_array + (0 .. NumOfVer-1));
    requires valid_graph: \valid_read(graph + (0 .. NumOfVer-1))
                          && \forall int i; 0 <= i < NumOfVer ==>
                          \valid_read(graph[i]+(0.. NumOfVer-1));
    requires valid_index:   0 <= sourceNode < NumOfVer;
    requires valid_distance: \forall int i; 0 <= i < NumOfVer ==>
                               node_array[i].distance == INT_MAX;
    requires valid_id:       \forall int i; 0 <= i < NumOfVer ==>
                               node_array[i].id == i;
    requires valid_parent:   \forall int i; 0 <= i < NumOfVer ==>
                               node_array[i].parent == NULL;

    assigns node_array[0 .. NumOfVer-1];

    ensures valid_array: \valid(node_array + (0 .. NumOfVer-1));

    ensures estimate_decrease: \forall int i;
                                0 <= i < NumOfVer
                                ==> node_array[i].distance
                                <= \old(node_array[i].distance);
*/
void dijkstra(  struct node node_array[NumOfVer],
                int graph[NumOfVer][NumOfVer], int sourceNode);
```

The next steps consist of propagating the implications of the ENSURE DECREASE, the clause at line 9 in Code listing 6.5, all the way to the ENSURE ESTIMATE_DECREASE. We will achieve this by adjusting the loop annotations and adding assertions to guide FRAMA-C in the verification process.

Because we cannot use the \OLD construct outside function contracts, we rely on the \AT construction with appropriate labels instead. The added assertions are as follows:

```
/*@ assert node_array[sourceNode].distance
    < \at(node_array[sourceNode].distance,Pre);
*/
/*@ assert \forall int g; 0 <= g < NumOfVer
    ==> node_array[g].distance
    <= \at(node_array[g].distance,Pre);
*/
/*@ assert \forall int g; 0 <= g < NumOfVer
    ==> node_array[g].distance
    == \at(node_array[g].distance,LoopCurrent);
*/
/*@ assert (node_array[i].distance
    < \at(node_array[i].distance,LoopCurrent));
*/
/*@ assert \forall int g; 0 <= g < NumOfVer
    ==> node_array[g].distance
    <= \at(node_array[g].distance,LoopCurrent);
*/
```

The first assertion is located after initializing the source node. The second assertion is after the insertion of every node into the array *set_Q*. The third assertion is in the main loop, right after the extraction of the node with the lowest shortest-path estimate. The fourth assertion is located after calling the `relax` function. The fifth assertion is just before the end of every iteration of the main loop.

To properly adjust the loop annotations, we introduce three new LOOP INVARIANTS. The first is an invariant of the main loop and states that after every iteration, the shortest-path estimate of every node is either lower or equal to its value before entering the loop.

The second loop invariant is added to the nested loop which attempts to relax every edge outgoing from the current node u. This invariant states: in each iteration, the shortest-path estimate of every node v incident to one of the edges already processed is either lower or equal to its value before entering the loop.

The third loop invariant is added to the same loop as the previous invariant. It states: the shortest-path estimate of every node v incident to an edge not yet processed by the loop is equal to its value before entering the loop.

See Code listing 6.16 for the adjusted loop annotations.

Code listing 6.16 Loop annotations with three new loop invariants

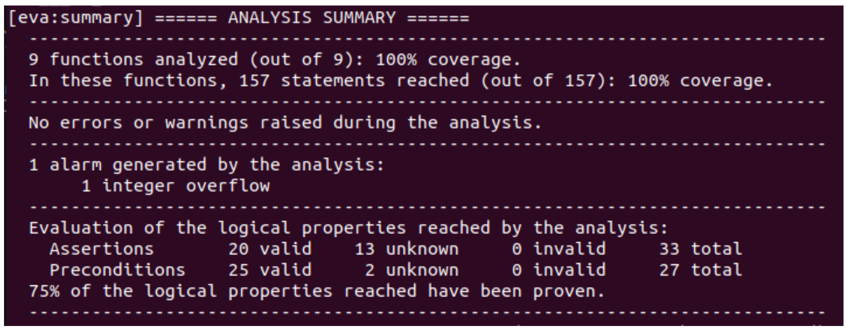
```
/*@
    loop invariant bound: 0 <= q_size <= NumOfVer;
    loop invariant q_size + s_size == NumOfVer;

    loop invariant \forall int k; 0 <= k < NumOfVer
                    ==> node_array[k].distance
                    <= \at(node_array[k].distance,LoopEntry);

    loop assigns    u, set_Q[0 .. NumOfVer-1],
                    q_size, s_size, node_array[0 .. NumOfVer-1];

    loop variant NumOfVer - s_size;
*/
for (int s_size = 0; s_size < NumOfVer; ++s_size){
    u = extract_min(set_Q, q_size);
    q_size--;
    /*@
        loop invariant bound:      0 <= i <= NumOfVer;

        loop invariant \forall int k; 0 <= k < i
                        ==> node_array[k].distance
                        <= \at(node_array[k].distance,LoopEntry);
        loop invariant \forall int k; i <= k < NumOfVer
                        ==> node_array[k].distance
                        == \at(node_array[k].distance,LoopEntry);

        loop assigns i, node_array[0 .. NumOfVer-1];
        loop variant end: NumOfVer-i;
    */
    for (int i = 0; i < NumOfVer; i++)
        if(static_graph[u->id][i] > 0 && u->distance
            < node_array[i].distance-static_graph[u->id][i])
            relax(u, &node_array[i], static_graph[u->id][i]);
}
```

Although the third loop invariant does not directly state anything about the potential decrease of shortest-path estimates, it is necessary. Without it `Frama-c` is unable to verify the other invariants. That is because the annotation *loop assigns node_array[0 .. NumOfVer-1];* signals to `Frama-c` that each iteration can modify the entire array, so without an invariant that expresses the state of that entire array in each iteration, `Frama-c` cannot know how exactly the state changes after every iteration.

6.5 Second Verification

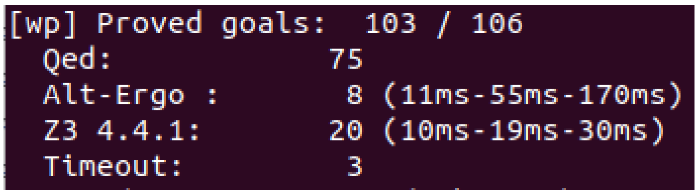
With the adjustments done, we will perform another analysis. Once again, we will use the `WP`, `RTE`, and `EVA` plugins, combined with solvers `Alt-Ergo` and `Z3`.

Figures 6.3, 6.4, and 6.5 showcase the output of the analysis.



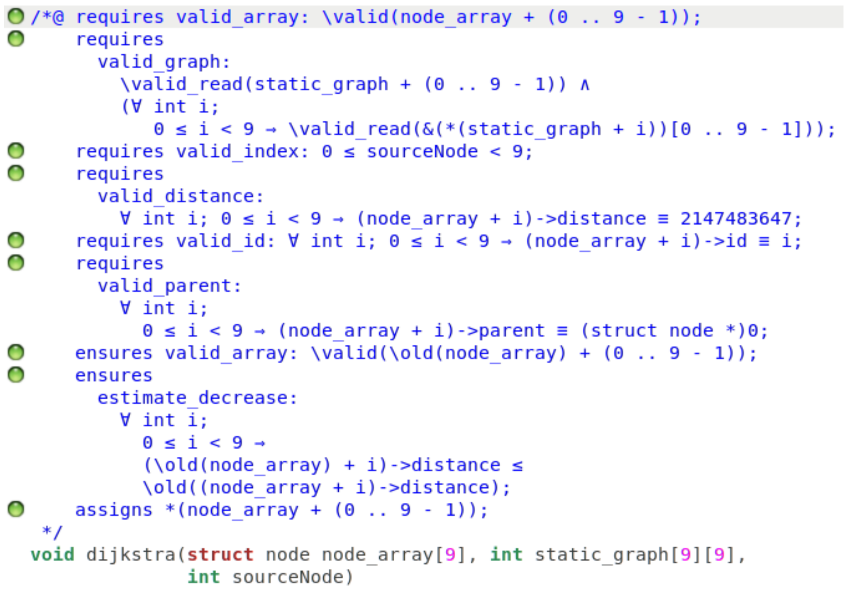
```
[eva:summary] ====== ANALYSIS SUMMARY ======
  ----------------------------------------------------------------------------
  9 functions analyzed (out of 9): 100% coverage.
  In these functions, 157 statements reached (out of 157): 100% coverage.
  ----------------------------------------------------------------------------
  No errors or warnings raised during the analysis.
  ----------------------------------------------------------------------------
  1 alarm generated by the analysis:
       1 integer overflow
  ----------------------------------------------------------------------------
  Evaluation of the logical properties reached by the analysis:
    Assertions        20 valid     13 unknown     0 invalid     33 total
    Preconditions     25 valid      2 unknown     0 invalid     27 total
  75% of the logical properties reached have been proven.
  ----------------------------------------------------------------------------
```

Figure 6.3 EVA summary of the second analysis. There are very few differences, as the added annotations are outside the purpose of the EVA plugin.



```
[wp] Proved goals:   103 / 106
  Qed:               75
  Alt-Ergo :          8 (11ms-55ms-170ms)
  Z3 4.4.1:          20 (10ms-19ms-30ms)
  Timeout:            3
```

Figure 6.4 WP summary of the second analysis. The new annotations created eleven new goals, eight of which had to be discharged to an external solver. The three goals that caused a timeout from the read_input function.



```
/*@ requires valid_array: \valid(node_array + (0 .. 9 - 1));
    requires
      valid_graph:
        \valid_read(static_graph + (0 .. 9 - 1)) ∧
        (∀ int i;
           0 ≤ i < 9 → \valid_read(&(*(static_graph + i))[0 .. 9 - 1]));
    requires valid_index: 0 ≤ sourceNode < 9;
    requires
      valid_distance:
        ∀ int i; 0 ≤ i < 9 → (node_array + i)->distance ≡ 2147483647;
    requires valid_id: ∀ int i; 0 ≤ i < 9 → (node_array + i)->id ≡ i;
    requires
      valid_parent:
        ∀ int i;
          0 ≤ i < 9 → (node_array + i)->parent ≡ (struct node *)0;
    ensures valid_array: \valid(\old(node_array) + (0 .. 9 - 1));
    ensures
      estimate_decrease:
        ∀ int i;
          0 ≤ i < 9 →
          (\old(node_array) + i)->distance ≤
          \old((node_array + i)->distance);
    assigns *(node_array + (0 .. 9 - 1));
 */
void dijkstra(struct node node_array[9], int static_graph[9][9],
              int sourceNode)
```

Figure 6.5 The function contract of function dijkstra showcasing the successful verification.

Chapter 7

Verification and security

As new software technologies are developed and the number of applications grows, so does the number of vulnerabilities in software. In 2023, over 29,000 new CVE numbers were issued for newly discovered vulnerabilities. This is nearly double the number from 2017 and almost five times that of 2016 [32].

These vulnerabilities pose various threats to both the software system and its users. Code security is therefore employed to reduce the volume and severity of the vulnerabilities. Traditional methods, such as static and dynamic analysis, code reviews, and penetration testing, serve an important purpose in identifying potential threats.

These methods are effective for discovering vulnerabilities that arise from:

- Configuration issues, such as incorrect settings in deployment environments. These issues are often detectable during runtime.
- Poor security practices, such as weak password policies or inadequate access control.
- Integration issues, where flaws occur from the interaction between different components, such as APIs.
- Flawed user interface design, including forms of cross-site scripting, which often arise from poor input validation.
- Race conditions, deadlocks, and performance limitations stemming from an inadequate system design, which can be discovered by dynamic analysis and stress tests.
- Flawed third-party libraries or dependencies.

However, some vulnerabilities and errors are better discovered by using formal verification. These include:

- Memory safety issues. A verified specification can guarantee exactly which parts of memory a program will manipulate, eliminating, for example, buffer overflow attacks. Furthermore, a verified implementation can guarantee that the program does not contain vulnerabilities such as use-after-free.
- Race conditions and deadlocks caused by improper implementation of multi-threaded programs.
- Arithmetic errors. Formal verification can prove that arithmetic operations cannot cause an overflow, underflow, or division by zero.

To showcase the importance of formal verification in information security we will introduce several notable examples of its real-world applications.

seL4 microkernel
The seL4 is a high-assurance, high-performance operating system microkernel [33]. It is unique because of its comprehensive and well-documented formal verification [34, 35] using the Isabelle automated theorem prover [36]. It is proven to enforce integrity and confidentiality, under certain assumptions.

OpenSSL HMAC
Researchers at Princeton University and Harvard University have proven, with Coq [14], that an OpenSSL [37] implementation of HMAC with SHA-256 correctly implements its functional specification and that these specifications guarantee the expected cryptographic properties [38].

Smart contracts
In the blockchain domain, formal verification is used to improve smart contract security. Smart contracts are a type of Ethereum [39] account but without a user. Instead, they are deployed and run as programmed. Formal verification is used to ensure that smart contracts are free from vulnerabilities and behave as intended [40].

Formal verification is a powerful tool in the development of secure applications, but it has disadvantages. The process of formal verification can be extremely time-consuming and complex. In addition, formal verification tools may not scale very well for large projects, leading the developers to verify only a portion of their work or forgo the process altogether.

Conclusion

The goals of this thesis were to create a study of Dijkstra's algorithm and formal verification through the use of the `Frama-c` platform, implement Dijkstra's algorithm in the C programming language and perform formal verification of this implementation.

The practical portion of this thesis began with an implementation of Dijkstra's algorithm using only static memory. The implementation was expanded by annotations written in the ANSI/ISO C Specification Language. Lastly, it was concluded by a successful verification of the implementation, with an emphasis on the absence of runtime errors. Furthermore, the implementation also satisfies an important property of the algorithm. To perform the formal verification, it used the `Frama-c` platform together with internal plugins `WP` and `EVA`.

In Chapter 1, the theoretical portion of the thesis introduces the reader to graph theory, the shortest path problem, and algorithms designed to solve it. Chapter 2 explains Dijkstra's algorithm and proves its correctness. Chapter 3 describes the `Frama-c` platform for formal verification of programs and shows its basic usage on examples. Chapter 4 serves as a brief introduction to Hoare logic and its rules. Chapter 5 describes the ANSI/ISO C specification language which was used in Chapter 6 to verify the implementation of Dijkstra's algorithm, thus fulfilling the main goal of this thesis. Finally, Chapter 7 discusses the various vulnerabilities and flaws that formal verification can effectively detect and its use in real-world examples.

This thesis can be expanded upon by implementing a verified data structure, such as a Fibonacci heap, to increase the efficiency of the implementation. Other properties of Dijkstra's algorithm could be verified as well, such as the correctness of the predecessor subgraph, or proving that the final calculated shortest-path estimate of each node is truly the shortest distance from the source.

Bibliography

1. CORMEN, Thomas H; LEISERSON, Charles E; RIVEST, Ronald L; STEIN, Clifford. *Introduction to algorithms*. MIT press, 2022.
2. RAMAKRISHNAN, KG; RODRIGUES, Manoel A. Optimal routing in shortest-path data networks. *Bell Labs Technical Journal*. 2001, vol. 6, no. 1, pp. 117–138.
3. KORF, Richard E. Optimal path-finding algorithms. In: *Search in artificial intelligence*. Springer, 1988, pp. 223–267.
4. LEE, Chi-Guhn; EPELMAN, Marina A; WHITE III, Chelsea C; BOZER, Yavuz A. A shortest path approach to the multiple-vehicle routing problem with split pick-ups. *Transportation research part B: Methodological*. 2006, vol. 40, no. 4, pp. 265–284.
5. WU, Changshan; MURRAY, Alan T. Optimizing public transit quality and system access: the multiple-route, maximal covering/shortest-path problem. *Environment and Planning B: Planning and Design*. 2005, vol. 32, no. 2, pp. 163–178.
6. CUI, Xiao; SHI, Hao. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*. 2011, vol. 11, no. 1, pp. 125–130.
7. FOEAD, Daniel; GHIFARI, Alifio; KUSUMA, Marchel Budi; HANAFIAH, Novita; GUNAWAN, Eric. A systematic literature review of A* pathfinding. *Procedia Computer Science*. 2021, vol. 179, pp. 507–514.
8. FREDMAN, Michael L; TARJAN, Robert Endre. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*. 1987, vol. 34, no. 3, pp. 596–615.
9. OCAMLPRO. OPAM (2.1.5) [software]. [`https://opam.ocaml.org/`]. 2024. Accessed: 2024-05-09.

10. DE MOURA, Leonardo; BJØRNER, Nikolaj. Z3: An efficient SMT solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
11. BARRETT, Clark; CONWAY, Christopher L; DETERS, Morgan; HADAREAN, Liana; JOVANOVIĆ, Dejan; KING, Tim; REYNOLDS, Andrew; TINELLI, Cesare. cvc4. In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 171–177.
12. *Génération Automatique de Preuves de Propriétés Arithmétiques* [online]. [visited on 2024-05-09]. Available from: `https://gappa.gitlabpages.inria.fr/`.
13. INTERNATIONAL, SRI. *Prototype Verification System (PVS)* [online]. [visited on 2024-05-09]. Available from: `https://pvs.csl.sri.com/`.
14. *The Coq proof assistant* [online]. [visited on 2024-05-16]. Available from: `https://coq.inria.fr/`.
15. BARRETT, Clark; TINELLI, Cesare. Satisfiability modulo theories. *Handbook of model checking*. 2018, pp. 305–343.
16. LRI. *Laboratoire de Recherche en Informatique* [online]. [visited on 2024-05-09]. Available from: `https://www.lri.fr/`.
17. WHY3 [online]. [visited on 2024-05-09]. Available from: `https://www.why3.org/`.
18. OCAMLPRO [online]. [visited on 2024-05-09]. Available from: `https://ocamlpro.com/`.
19. SPARK [online]. [visited on 2024-05-09]. Available from: `https://www.adacore.com/sparkpro`.
20. B-METHOD [online]. [visited on 2024-05-09]. Available from: `https://www.atelierb.eu/en/`.
21. OCAMLPRO. *About Alt-Ergo* [online]. [visited on 2024-05-09]. Available from: `https://alt-ergo.ocamlpro.com/`.
22. *Z3 solver, open source.* [online]. [visited on 2024-05-09]. Available from: `https://github.com/Z3Prover/z3`.
23. *JavaScript runtime environment.* [online]. [visited on 2024-05-09]. Available from: `https://nodejs.org/en`.
24. *Cross-platfrom desktop apps.* [online]. [visited on 2024-05-09]. Available from: `https://www.electronjs.org/`.
25. *TypeScript.* [online]. [visited on 2024-05-09]. Available from: `https://www.typescriptlang.org/`.
26. HOARE, Charles Antony Richard. An axiomatic basis for computer programming. *Communications of the ACM*. 1969, vol. 12, no. 10, pp. 576–580.

27. FLOYD, Robert W. Assigning meanings to programs. In: SCHWARTZ, J. T. (ed.). *Proc. Symposium on Applied Mathematics*. Providence, RI: American Mathematical Society, 1967, vol. 19, pp. 19–32. Mathematical Aspects of Computer Science.

28. GERLACH, Jens; EFREMOV, Denis; SIKATZKI, Tim. *ACSL by Example* [`https://github.com/fraunhoferfokus/acsl-by-example/`]. 2024. A fairly complete tour of ACSL and WP features through various functions inspired from C++ STL [Source code].

29. *ACSl - Frama-c* [online]. [visited on 2024-05-09]. Available from: `https://frama-c.com/html/acsl.html`.

30. BAUDIN, Patrick; CUOQ, Pascal; FILLIÂTRE, Jean-Christophe; MARCHÉ, Claude; MONATE, Benjamin; MOY, Yannick; PREVOSTO, Virgile. *ACSL: ANSI/ISO C Specification Language: Version 1.20* [online]. [visited on 2024-05-10]. Available from: `https://frama-c.com/download/acsl.pdf`.

31. HATCLIFF, John; LEAVENS, Gary T; LEINO, K Rustan M; MÜLLER, Peter; PARKINSON, Matthew. Behavioral interface specification languages. *ACM Computing Surveys (CSUR)*. 2012, vol. 44, no. 3, pp. 1–58.

32. *Number of CVEs by year* [online]. [visited on 2024-05-16]. Available from: `https://www.cvedetails.com/browse-by-date.php`.

33. *The seL4 Microkernel* [online]. [visited on 2024-05-16]. Available from: `https://sel4.systems/`.

34. KLEIN, Gerwin; ELPHINSTONE, Kevin; HEISER, Gernot; ANDRONICK, June; COCK, David; DERRIN, Philip; ELKADUWE, Dhammika; ENGELHARDT, Kai; KOLANSKI, Rafal; NORRISH, Michael, et al. seL4: Formal verification of an OS kernel. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

35. KLEIN, Gerwin; ANDRONICK, June; ELPHINSTONE, Kevin; MURRAY, Toby; SEWELL, Thomas; KOLANSKI, Rafal; HEISER, Gernot. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)*. 2014, vol. 32, no. 1, pp. 1–70.

36. *Isabelle proof assistant* [online]. [visited on 2024-05-16]. Available from: `https://isabelle.in.tum.de/`.

37. *OpenSSL* [online]. [visited on 2024-05-16]. Available from: `https://www.openssl.org/`.

38. BERINGER, Lennart; PETCHER, Adam; YE, Katherine Q.; APPEL, Andrew W. Verified Correctness and Security of OpenSSL HMAC. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 207–221. ISBN 978-1-939133-11-3. Available also from: `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer`.

39. *Ethereum technology* [online]. [visited on 2024-05-16]. Available from: `https://ethereum.org/en/`.

40. BHARGAVAN, Karthikeyan; DELIGNAT-LAVAUD, Antoine; FOURNET, Cédric; GOLLAMUDI, Anitha; GONTHIER, Georges; KOBEISSI, Nadim; KULATOVA, Natalia; RASTOGI, Aseem; SIBUT-PINOTE, Thomas; SWAMY, Nikhil, et al. Formal verification of smart contracts: Short paper. In: *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*. 2016, pp. 91–96.

Contents of the attachment

```
kupsaja1-thesis.pdf ................ compiled thesis in the PDF format
src
  main.c ..................... source file for running Dijkstra's algorithm
  dijkstra.h ........................ header file of Dijkstra's algorithm
  dijkstra.c ........................ source file of Dijkstra's algorithm
thesis .......................................... the thesis LaTeX source
```