Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A Framework for the Verification of Certifying Computations

Eyad Alkassar · Sascha Böhme · Kurt Mehlhorn · Christine Rizkallah

the date of receipt and acceptance should be inserted later

Abstract Formal verification of complex algorithms is challenging. Verifying their implementations goes beyond the state of the art of current verification tools and proving their correctness usually involves intricate mathematical theorems. Certifying algorithms compute in addition to each output a witness certifying that the output is correct. A checker for such a witness is usually much simpler than the original algorithm – yet it is all the user has to trust. Verification of checkers is feasible with current tools and leads to computations that can be completely trusted. We describe a framework to seamlessly verify certifying computations. We use the automatic verifier VCC for establishing correctness of the checker, and the interactive theorem prover Isabelle/HOL for high-level mathematical properties of algorithms. We demonstrate the effectiveness of our approach by presenting the verification of typical examples of the industrial-level and widespread algorithmic library LEDA.

1 Introduction

One of the most prominent and costly problems in software engineering is correctness of software. In this article, we are concerned with software for difficult algorithmic problems, e.g., in the domain of graphs. The algorithms for such problems are complex; formal verification of the resulting programs is beyond the state of the art. We show how to obtain *formal instance correctness*, i.e., formal proofs that outputs for particular inputs are correct. We do so by combining the concept of certifying algorithms with methods for code verification and theorem proving.

A *certifying algorithm* [6,36,23] produces with each output a *certificate* or *witness* that the *particular output* is correct. The accompanying *checker* inspects the witness

E. Alkassar
Fachbereich Informatik, Universität des Saarlandes, Germany

S. Böhme
Institut für Informatik, Technische Universität München, Germany

K. Mehlhorn · C. Rizkallah
Max-Planck-Institut für Informatik, Saarbrücken, Germany

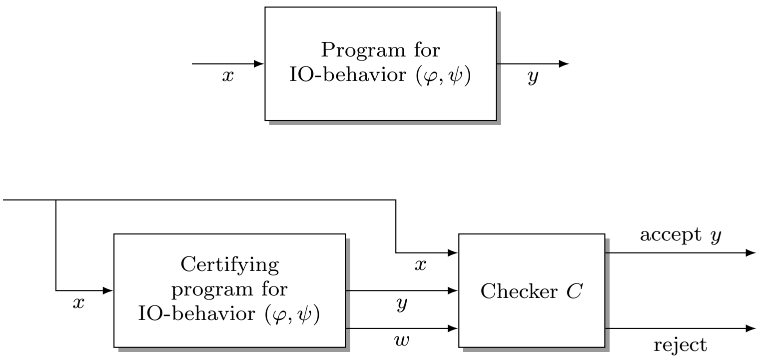


Fig. 1: The top figure shows the I/O behavior of a conventional program for IO-behavior (φ, ψ). The user feeds an input x satisfying $\varphi(x)$ to the program and the program returns an output y satisfying $\psi(x, y)$. A certifying algorithm for IO-behavior (φ, ψ) computes y and a witness w. The checker C accepts the triple (x, y, w) if and only if w is a valid witness for the postcondition $\psi(x, y)$, i.e., proves $\psi(x, y)$.

and accepts it if the witness proves that y is a correct output for input x. Otherwise, the checker rejects the output or witness as buggy. Figure 1 contrasts a standard algorithm with a certifying algorithm for IO-behavior (φ, ψ). An algorithm for IO-behavior (φ, ψ) receives an input x satisfying a precondition $\varphi(x)$ and is supposed to deliver an output y satisfying the postcondition $\psi(x, y)$. We call such a y a *correct output*. If the input does not satisfy the precondition, the result of the computation is unspecified. A user of a standard algorithm has, in general, no means to know that y is a correct output and has not been compromised by a bug.

A certifying algorithm with input x outputs y and a witness w. The triple (x, y, w) is then fed to the accompanying checker. The checker accepts the triple if w proves that y is a correct output for x. Otherwise, the checker rejects. If the checker accepts, the user may proceed with complete confidence that output y has not been compromised by a bug. If the checker rejects, either y is incorrect or w is not a proof of the correctness of y.

Certifying algorithms are a key design principle of the algorithmic library LEDA [24]: Checkers are an integral part of the library and may (optionally) be invoked after every execution of a LEDA algorithm. Adoption of the principle greatly improved the reliability of the library. However, how can one be sure that the checker programs are correct? The third author used to answer: "Checkers are simple programs with little algorithmic complexity. Hence one may assume that their implementations are correct."

We take the certifying-algorithms approach a step further by developing a methodology for the verification of certifying computations. We demonstrate it on three examples: connectivity of graphs, single-source shortest paths in graphs with nonnegative edge weights, and maximum cardinality matching in graphs. The latter is one of the more complex algorithms in LEDA. The description of the algorithm and its implementation in [24] comprises 15 pages. In contrast, the checker is less than a page. Our formalization revealed that the checker program in LEDA is incomplete.

We introduce our methodology in Section 2 and give detailed case studies in Section 4 before evaluating our approach and the obtained results in Section 5. In Section 3 we survey the verification tools VCC and Isabelle/HOL. Section 6 discusses related work and Section 7 offers conclusions. The companion web page `http://www21.in.tum.de/~boehmes/certifying_computations.html` contains additional material, in particular, the program listings including VCC annotations and the Isabelle/HOL proofs.

This article is a revised and extended version of a paper published by the same authors at CAV 2011 [1]. We added two case studies to underline the feasibility and elegance of our approach. Moreover, we strengthened and simplified our approach. We now prove total correctness of the checkers and not only partial correctness, and we establish that the checker accepts a triple (x, y, w) if and only if w is a valid witness for output y. Previously, we only proved soundness and not completeness. The simplification results from dropping concrete specifications in VCC. This goes along with more succinct specifications and proofs for our original case study thanks to a new version of VCC. We modified our original case study accordingly.

2 Outline of Methodology

We consider algorithms taking an input from a set X and producing an output in a set Y and a witness in a set W. The input $x \in X$ is supposed to satisfy a precondition $\varphi(x)$ and the input together with the output $y \in Y$ is supposed to satisfy a postcondition $\psi(x, y)$. A *witness predicate* for a specification with precondition φ and postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$, where W is a set of witnesses, with the following *witness property*:

$$\varphi(x) \wedge \mathcal{W}(x, y, w) \longrightarrow \psi(x, y) \qquad (1)$$

In contrast to algorithms which work on abstract sets X, Y, and W, programs as their implementations operate on concrete representations of abstract objects. We use $\overline{X}$, $\overline{Y}$, and $\overline{W}$ for the set of representations of objects in X, Y, and W, respectively and assume mappings $i_X : \overline{X} \to X$, $i_Y : \overline{Y} \to Y$, and $i_W : \overline{W} \to W$. The checker program C receives a triple $(\overline{x}, \overline{y}, \overline{w})$ and is supposed to check whether it fulfils the witness property. More precisely, let $x = i_X(\overline{x})$, $y = i_Y(\overline{y})$, and $w = i_W(\overline{w})$. If $\neg\varphi(x)$, C may do anything (run forever or halt with an arbitrary output). If $\varphi(x)$, C must halt and either accept or reject. It is supposed to accept if $\mathcal{W}(x, y, w)$ holds and supposed to reject otherwise. The following proof obligations arise:

Checker Correctness: A proof that C checks the witness predicate assuming that the precondition[1] holds, i.e., on input $(\overline{x}, \overline{y}, \overline{w})$ and with $x = i_X(\overline{x})$, $y = i_Y(\overline{y})$, and $w = i_W(\overline{w})$:

1. If $\varphi(x)$, C halts.
2. If $\varphi(x)$ and $\mathcal{W}(x, y, w)$, C accepts, and if $\varphi(x)$ and $\neg\mathcal{W}(x, y, w)$, C rejects. In other words, if $\varphi(x)$, C accepts if and only if $\mathcal{W}(x, y, w)$, and C rejects if and only if $\neg\mathcal{W}(x, y, w)$.

[1] We stress that the checker has the same precondition as the algorithm.

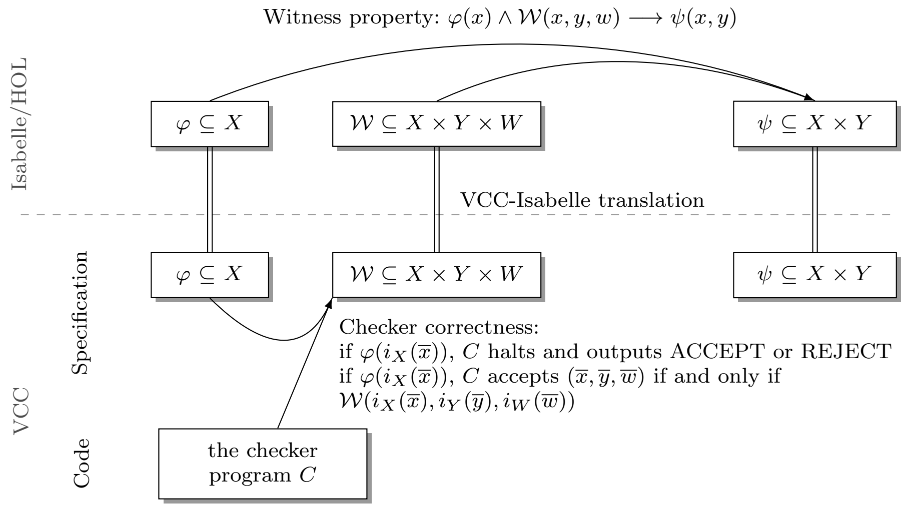


Fig. 2: Verification Framework

Witness Property: A proof for the implication (1).

Theorem 1 *Assume that the proof obligations are fulfilled. Let* $(\overline{x}, \overline{y}, \overline{w}) \in \overline{X} \times \overline{Y} \times \overline{W}$ *and let* $x = i_X(\overline{x})$, $y = i_Y(\overline{y})$, *and* $w = i_W(\overline{w})$.

If C *accepts a triple* $(\overline{x}, \overline{y}, \overline{w})$, $\varphi(x) \longrightarrow \psi(x, y)$ *by a formal proof. If* C *rejects a triple* $(\overline{x}, \overline{y}, \overline{w})$, $\neg\varphi(x) \vee \neg\mathcal{W}(x, y, w)$ *by a formal proof.*

Proof If C accepts $(\overline{x}, \overline{y}, \overline{w})$, we have $\varphi(x) \longrightarrow \mathcal{W}(x, y, w)$ by the correctness proof of C. Then by (1) we have a formal proof for $\varphi(x) \longrightarrow \psi(x, y)$. Conversely, if C rejects the triple, the correctness proof of C establishes $\neg\varphi(x) \vee \neg\mathcal{W}(x, y, w)$. □

The checker programs for our three examples compute total functions as they use only for-loops for iterations; they use neither while-loops nor recursion. We will give formal termination proofs. If x violates the precondition, the checkers may give and generally will give nonsensical output or halt with a run-time error.

We next discuss how we fulfill the proof obligations in a *comprehensive* and *efficient* framework, see Figure 2. Comprehensive means that the final proof formally combines (as much as possible at the syntactic level) the correctness arguments for all levels (implementation, abstraction and mathematical theory). Efficient means to use the right tool for the right task. For example, applying a general theorem prover to verify imperative code would involve a lot of language-specific overhead and lead to less automation; similarly, a specialized code verifier is often not powerful enough to cover nontrivial mathematical properties. The aims comprehensiveness and efficiency seem to be conflicting, as different tools usually come with different languages, axiomatization sets, etc. Our solution is to use second-order logic as a common interface language.

LEDA is written in C++ [24]. Our aim is to verify code which is as close as possible to the original implementation; by this we demonstrate the feasibility of verifying already established libraries written in imperative languages such as C. Thus we verify code with VCC [14], an automatic code verifier for full C. Our choice is motivated by the maturity of the tool and the provision of an assertion language which is rich enough for our requirements. In the Verisoft XT project [38] VCC was successfully used to verify tens of thousands of lines of C code. It offers an assertion language with ghost code and types such as maps and unbounded integers. This gives enough expressiveness to quantify over graphs, labelings, etc. and simplifies the translation to other proof systems. For verifying the mathematical part, we resort to Isabelle/HOL, a higher-order-logic interactive theorem prover [26], because of the large amount of already formalized mathematics, its descriptive proof format and its various automatic proof methods and tools. In Section 3 we review both systems. Figure 2 shows the workflow for verifying checkers.

Checker Verification: Starting point is the checker code written in C. Using VCC we annotate the functions and data structures, such that the witness predicate $\mathcal{W}$ can be established as postcondition of the checker function. We define the witness predicate and the pre- and postcondition as well as the mappings i_X, i_Y and i_W as pure mathematical objects using VCC ghost types and ghost functions.

Export to Isabelle/HOL: Establishing the witness property involves in general nontrivial mathematical reasoning. We therefore translate the precondition, witness predicate, postcondition, and the abstract representations of the input, output, and witness from VCC to Isabelle/HOL. Since we formulated them as pure mathematical objects in VCC, this translation is purely syntactical and does not involve any VCC specifics. The translation could easily be automated.

Witness Property: We prove the witness property using Isabelle/HOL. It is convenient to formulate this theorem on yet a higher level of abstraction and provide linking proofs to connect the exported VCC predicates with their abstracted counterparts.

We stress that the overall correctness theorem, i.e., the witness property, can be formulated in VCC; this is important for usability. A user of a verified checker only has to look at its VCC specification; the fact that we outsource the proof of the witness property to Isabelle/HOL is of no concern to him. We formulate the witness property as an axiom in VCC. This is sound since we restrict the language for describing the witness property to second-order logic which guarantees that we can express it equivalently in Isabelle's higher-order logic (cf. Section 3). More precisely, since the VCC formulation of the witness property is valid if and only if its translation to Isabelle is valid and since Isabelle is consistent and hence only valid statements can be proven, it is sound to add the witness property as an axiom to VCC.

The reader may wonder, why we do not formally prove the existence of a witness:

$$\forall x\, y.\ \varphi(x) \land \psi(x, y) \longrightarrow \exists w.\ \mathcal{W}(x, y, w).$$

The existence of a witness is part of the correctness argument of the solution algorithm (the shortest-path algorithm, the maximum-matching algorithm) which we do not formalize. Rather, the execution of the solution algorithm establishes the existence of a witness whenever it is called for a specific input $\overline{x}$. It returns $\overline{y}$

and $\overline{w}$ which we then hand to the checker C. In this way, we obtain formal instance correctness without having to verify the solution algorithm. Of course, this leaves the possibility that the solution algorithm is incorrect and does not always provide a $\overline{y}$ and $\overline{w}$ such that the checker accepts $(\overline{x}, \overline{y}, \overline{w})$.

For a user, the checker is what counts. The user can trust it, because it has been formally verified. Moreover, if it accepts a triple $(\overline{x}, \overline{y}, \overline{w})$, the user can be sure, that y is a correct output provided that x satisfied the precondition of the algorithm. This is because the witness property has been formally verified. If the checker rejects a triple the user knows that either x does not satisfy the precondition or (x, y, w) does not satisfy the witness predicate. The method by which $\overline{y}$ and $\overline{w}$ were produced is of no concern for the user.

The witness property is formulated with respect to a certain IO-behavior (φ, ψ) and not with respect to a particular algorithm realizing the IO-behavior. Therefore a checker can be used in connection with any certifying algorithm for IO-behavior (φ, ψ) that produces the appropriate witnesses.

3 Tool Overview: VCC and Isabelle/HOL

VCC [14] is an assertional, automatic, deductive code verifier for full C code. Specifications in the form of function contracts, data invariants and loop invariants as well as further annotations, e.g., to maintain inductively defined information or to guide VCC otherwise, are added directly into the C source code. During regular build, these annotations are ignored. From the annotated program, VCC generates verification conditions for (partial and/or total) correctness, which it then tries to discharge using the Boogie verifier [3] and the automatic theorem prover Z3 [25].

Verification in VCC makes heavy use of ghost data and code, enclosed by _(and), used for reasoning about the program but omitted from the concrete implementation. VCC provides ghost objects, ghost fields of structured data types, local ghost variables, ghost function parameters, and ghost code. Ghosts can not only use C data types but also additional mathematical data types, e.g., mathematical integers (**integer**) and natural numbers (**natural**), records (similar to C structures) and maps (with syntax similar to C arrays). VCC ensures that information does not flow from ghost state to non-ghost state, and that all ghost code terminates; these checks guarantee that program execution when projected to non-ghost code is not affected by ghost code.

Isabelle/HOL [26] is an interactive theorem prover for classical higher-order logic based on Church's simply-typed lambda calculus. Internally, the system is built on top of an inference kernel which provides only a small number of rules to construct theorems; complex deductions (especially by automatic proof methods) ultimately rely on these rules only. This approach, called LCF due to its pioneering system [18], guarantees correctness as long as the inference kernel is correct. Isabelle/HOL comes with a rich set of already formalized theories, among which are natural numbers and integers as well as sets, finite sets and—as a recent addition [30]—graphs. New types can also be introduced by defining them as records (isomorphic to tuples with named update and selector functions), among other means. New constants can be introduced, for example, via definitions relative to already existing constants.

Proofs in Isabelle/HOL can be written in a style close to that of mathematical textbooks called Isabelle/Isar. The user structures the proof and the system fills in the gaps by its automatic proof methods. Moreover, one can use locales which provide a method for defining local scopes in which constants are defined and assumptions about them are made. Theorems can be proven in the context of a locale and can use the constants and depend on the assumptions of this locale. A locale can be instantiated to concrete entities if the user is able to show that the latter fulfill the locale assumptions.

We restrict ourselves to a subset of VCC's specification language for the types and propositions that we pass from VCC to Isabelle. Simple types are natural numbers, integers, algebraic datatypes over simple types, and ghost records whose fields are simple types. Rich types are simple types, ghost records whose fields are rich types, and mappings from simple types to rich types. Propositions can be formed by usual logical connectives, quantifiers over variables of rich types, arithmetic expressions, equalities, and user-defined pure, stateless functions whose argument and result types are rich and whose definitions or contracts are again propositions, possibly using pattern matching over algebraic datatypes. This subset of VCC's language is a second-order logic. Any type or function of this logic can be expressed equivalently in Isabelle/HOL, essentially by syntactic rewriting. More precisely, VCC algebraic datatypes can be translated into Isabelle datatypes, VCC ghost records can be translated into Isabelle records, and pure VCC ghost functions can be translated into Isabelle function definitions. The former two translations are sound and complete because the semantics of datatypes and records is the same in both systems; the latter is sound and complete as VCC's underlying logic is subsumed by the higher-order logic of Isabelle/HOL. The translation maps VCC specification types (**bool**, **natural**, **integer**, and map types) to equivalent Isabelle types (*bool*, *nat*, *int*, and function types) and maps VCC expressions comprising logical connectives, quantifiers, arithmetic operations, equality and specification functions to equivalent Isabelle terms.

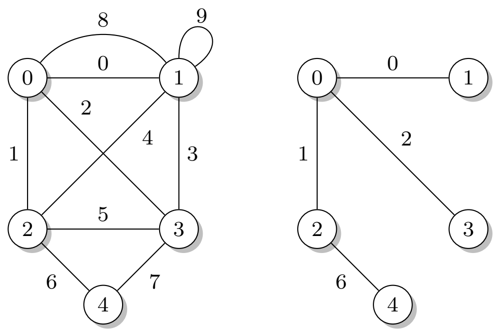
4 Case Studies

We present three case studies from the domain of graphs. We obtain formal instance correctness. Our starting point are the certifying algorithms and the corresponding checkers in LEDA. We give formal proofs for the correctness of the checkers, for the related witness properties, and the connection between them. Except for the witness property, which is proven in Isabelle/HOL, all presented abstractions and functions have been formally verified using VCC.

All files related to our formalization can be obtained from the following URL: http://www21.in.tum.de/~boehmes/certifying_computations.html

4.1 Case Study: Connected Components in Graphs

Given an undirected graph $G = (V, E)$, we consider an algorithm that decides whether G is connected, i.e., there is a path between any pair of vertices [24, Section 7.4]. In the negative case, i.e., when the graph is not connected, there is a simple certificate. It consists of a cut S, i.e., a nonempty subset S of the vertices



(a) A connected graph G (b) A spanning tree of G

vertex	0	1	2	3	4
parent-edge	$\perp$	0	1	2	6
num	0	1	1	1	2

(c) Spanning tree representation

Fig. 3: An example of a connected graph G and a spanning tree of G witnessing its connectivity. The vertices belong to the set $\{0, \ldots, n-1\}$ and the edges are pairs of vertices indexed by an identifier ranging from 0 to $m-1$, where n and m are the number of vertices and edges in G. The spanning tree in (b) can be represented by a root vertex $r = 0$ and functions *parent-edge* and *num* as shown in the table in (c). Graphs may have self loops and parallel edges.

with $S \neq V$, such that every edge of the graph has either both or no endpoint in S. In other words, no edge crosses the cut. In the positive case, i.e., when the given graph is connected, the algorithm can produce as certificate a spanning tree of G. A spanning tree of G is a subgraph of G which is a tree and contains all vertices of G. We concentrate here on the more complicated positive case. We describe a checker for the spanning tree certificate and the verification of this checker. On a high level, we instantiate our general approach as follows:

$$
\begin{array}{rcl}
\text{input } x & = & \text{an undirected graph } G = (V, E) \\
\text{output } y & = & \text{either } \mathit{True} \text{ or } \mathit{False} \text{ indicating whether } G \text{ is connected} \\
\text{witness } w & = & \text{a cut or a spanning tree} \\
\varphi(x) & = & G \text{ is wellformed, i.e., } E \subseteq V \times V,\ V \text{ and } E \text{ are finite sets} \\
\mathcal{W}(x, y, w) & = & y \text{ is } \mathit{True} \text{ and } w \text{ is a spanning tree of } G \text{ or } y \text{ is } \mathit{False} \text{ and } w \text{ is} \\
 & & \text{a cut} \\
\psi(x, y) & = & \text{if } y \text{ is } \mathit{True},\ G \text{ is connected and if } y \text{ is } \mathit{False},\ G \text{ is not con-} \\
 & & \text{nected.}
\end{array}
$$

We restrict ourselves to the positive case $y = \mathit{True}$. Figure 3 shows a graph G and a spanning tree of it. We represent spanning trees by functions *parent-edge* and *num* and a root vertex r and view the edges of the tree oriented towards r: for $v \neq r$, *parent-edge*(v) is the first edge on the path from v to r, *parent-edge*$(r) = \perp$, and *num*(v) is the length of the path from v to r for all v. We present the implementation of a checker in Section 4.1.1, detail the formalization of the witness predicate and the verification of the checker in Section 4.1.2 and prove the witness property in Section 4.1.3. The relevant files in the companion website are check_connected.c (C-code and checker correctness), check_connected.thy (Isabelle/HOL representation of the VCC checker specification and witness property), ConnectedComponents.thy (abstract verification of the witness property in a locale), and ConnectedComponents_Link.thy (instantiation of the locale with the Isabelle/HOL representation of the VCC specification).

```
typedef unsigned Nat;
typedef Nat Vertex;
typedef Nat Edge_Id;
typedef struct { Nat s; Nat t; } Edge;
typedef struct { Nat m; Nat n; Edge∗ es; } Graph;
```

Listing 1: A representation of graphs in C. The field m gives the number of edges (and hence the length of the array es) and n gives the maximum number of vertices in the graph.

4.1.1 Connected-Components Checker

We begin by fixing a representation of graphs in the programming language C, see Listing 1. Vertices are taken from the set $\{0, \ldots, \mathsf{n}-1\}$. Edges are pairs where the first vertex is labeled s (for source) and the second vertex is labeled t (for target). Edges are stored in an array es which is indexed by edge identifiers that range from 0 to $\mathsf{m}-1$. We require that the two vertices of each edge belong to the graph, i.e., that they are from the range $\{0, \ldots, \mathsf{n}-1\}$, and call graphs with this property *wellformed*. We use the same data structure for directed and undirected graphs. For directed graphs, an edge e with $\mathsf{e.s} = \mathsf{u}$ and $\mathsf{e.t} = \mathsf{v}$ is directed from u to v. For undirected graphs, it represents the unordered pair {u,v}.

We represent spanning trees as explained before. Instead of functions, we use two arrays parent_edge and num in addition to a root vertex r. The parent_edge array maps r to a negative value, i.e., to a value that does not identify any edge (recall that edges are identified by nonnegative integers).

The connected-graph checker is a function that succeeds if the two functions check_r and check_parent_num (Listing 2) succeed. The first function checks that r is indeed the root of the spanning tree. The second function checks for every vertex v different from r that the edge parent_edge[v] is incident to v and that the other endpoint of the edge has a number which is one smaller than num[v].

4.1.2 Checker Correctness

We need to provide abstract representations for graphs and paths. Our decision was to keep them close to the concrete representation for two reasons. It makes detecting differences and hence potential bugs easier for the programmer. It also makes reasoning for VCC simpler. The declaration of abstract graphs is given in Listing 3 together with the ghost predicate \wellformed for describing when an abstract graph is wellformed. This ghost predicate plays the role of the precondition φ in this case study. Our abstract version of the num array is a mapping from vertices to natural numbers. The abstract version of the parent_edge array is a mapping from vertices to the set $\mathbb{N} \cup \{\perp\}$; we use $\perp$ to model an undefined value. To represent this set, we define an algebraic datatype Option:

```
_(datatype \Option
{
    case \none();
    case \some(\Edge_Id e);
})
```

```
int check_r(Graph* G, Vertex r, int* parent_edge, Nat* num)
{
    return r < G->n && num[r] == 0 && parent_edge[r] == -1;
}

int check_parent_num(Graph* G, Vertex r, int* parent_edge, Nat* num)
{
    Vertex v, a, b; Edge_Id e;

    for (v = 0; v < G->n; v++)
    {
        if (v == r) continue;

        if (parent_edge[v] < 0 || ((Edge_Id)parent_edge[v]) >= G->m) return FALSE;

        e = (Edge_Id)parent_edge[v];
        a = G->es[e].s;
        b = G->es[e].t;

        if (v == a && num[v] == num[b] + 1) continue;
        if (v == b && num[v] == num[a] + 1) continue;
        return FALSE;
    }
    return TRUE;
}
```

Listing 2: Two functions comprising the connected-components checker

```
_(typedef \natural \Vertex)
_(typedef \natural \Edge_Id)
_(record \Edge {
    \Vertex src;
    \Vertex trg;
})
_(record \Graph {
    \natural num_verts;
    \natural num_edges;
    \Edge edge[\Edge_Id];
})
```

```
_(def \bool \wellformed(\Graph G)
{
    return
        ∀ \Edge_Id i; i < G.num_edges ⟶
            G.edge[i].src < G.num_verts ∧
            G.edge[i].trg < G.num_verts;
})
```

Listing 3: Abstract graphs and a predicate to describe wellformed graphs

with operations \is_some(o) for the test $o \neq \bot$ and \the(o) for extracting an edge identifier. The abstraction functions that map concrete data to pure mathematical data are straightforward to define. For example,

```
_(def \Graph \abs_graph(Graph* G)
{
    return (\Graph) {
        .num_verts = G->n,
        .num_edges = G->m,
        .edge =
            \lambda \Edge_Id i;
                (i < G->m) ?
                    (\Edge) { .src = G->es[i].s, .trg = G->es[i].t } :
                    (\Edge) { .src = 0, .trg = 0 }};
```

```
})
```

abstracts a concrete graph G into an abstract graph of type \Graph. Using the abstract types we define the witness predicate as a conjunction of two properties, one for each of the checker functions in Listing 2.

check_r Vertex r is the root of the spanning tree:

```
r < G.num_verts ∧ ¬\is_some(parent_edge[r]) ∧ num[r] = 0
```

check_parent_num Every vertex of the graph is connected to some other vertex which is closer to r:

```
∀ \Vertex v; v < G.num_verts ∧ v ≠ r ⟶
    \is_some(parent_edge[v]) ∧ \the(parent_edge[v]) < G.num_edges ∧
    (G.edge[\the(parent_edge[v])].trg == v ∧
      num[v] == num[G.edge[\the(parent_edge[v])].src] + 1 ∨
     G.edge[\the(parent_edge[v])].src == v ∧
      num[v] == num[G.edge[\the(parent_edge[v])].trg] + 1)
```

Thanks to the low level of abstraction in the above predicates, the two checker functions are easily verified. For the verification of check_parent_num we need to annotate the loop with the second conjunct above where G.num_verts is replaced by the loop variable as loop invariant. Moreover, for every **return** FALSE we need to assert, or restate, on the abstract level the properties that are violated to guide VCC. Otherwise, it would fail to show completeness of the checker. For instance,

```
if (parent_edge[v] < 0 || ((Edge_Id)parent_edge[v]) >= G−>m)
{
    _(assert ¬\is_some(P[v]) ∨ \the(P[v]) ≥ \abs_graph(G).num_edges)
    return FALSE;
}
```

is one of the two occurrences of such extra assertions in check_parent_num.

We express the postcondition of the checker, i.e., that any pair of vertices of the graph G is connected by a path, as follows:

```
∀ \Vertex u, v; u < G.num_verts ∧ v < G.num_verts ⟶
    ∃ \Path p; \natural n; \is_path(G, p, n, u, v);
```

where the type \Path is a sequence of vertices, represented as a mapping from natural numbers to vertices, and where the predicate \is_path(G, p, n, u, v) characterizes that path p of length n starts at u, ends at v and contains pairwise distinct vertices only that are connected by edges of the graph:

```
p[0] == u ∧
p[n] == v ∧
(∀ \natural i; i ≤ n ⟶ p[i] < G.num_verts) ∧
(∀ \natural i; i < n ⟶ \is_edge(G, p[i], p[i+1])) ∧
(∀ \natural i, j; i ≤ n ∧ j ≤ n ∧ i ≠ j ⟶ p[i] ≠ p[j])
```

The predicate \is_edge(G, u, v), for any two vertices u and v of G, is true if and only if u and v are the endpoints of an edge of G:

```
∃ \Edge_Id i; i < G.num_edges ∧
    (G.edge[i].src = u ∧ G.edge[i].trg = v ∨
     G.edge[i].src = v ∧ G.edge[i].trg = u)
```

We give a formal proof for the implication from precondition and witness predicate to the postcondition in the next section.

locale *connected-components-locale* = *pseudo-digraph* +
  fixes *num* :: $\alpha \Rightarrow$ *nat*
  fixes *parent-edge* :: $\alpha \Rightarrow \beta$ *option*
  fixes r :: α
  assumes r_assms: $r \in$ *verts* $G \wedge$ *parent-edge* r = *None* $\wedge$ *num* $r = 0$
  assumes parent_num_assms:
    $\bigwedge v.\ v \in$ *verts* $G \wedge v \neq r \Longrightarrow$
      $\exists\, e \in$ *edges* G.
        parent-edge v = *Some* e $\wedge$
        target $G\ e = v \wedge$
        num v = *num* (*start* $G\ e$) + 1

Listing 4: Locale for the connected-components proof in Isabelle

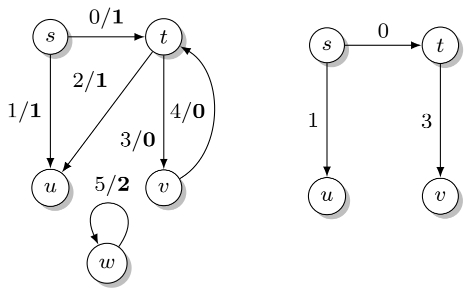
4.1.3 Proof of the Witness Property for the Connected-Components Checker

We prove in Isabelle that a spanning tree witnesses connectivity of a graph. We do so in two steps. First, we perform a high-level proof where we abstract from concrete representations of graphs and spanning trees. We then instantiate this proof with the data structures and according properties exported from VCC.

Our formalization builds on the Isabelle graph library developed by Lars Noschinski [30]. Graphs in this library are directed. A *pseudo-digraph* is a wellformed directed graph with a finite set of vertices and a finite set of edges; the library reserves the word *digraph* for graphs without parallel edges and self-loops. We represent undirected graphs as bidirected graphs, i.e., directed graphs containing for every edge (u, v) also the reversed edge (v, u). The function *mk-symmetric* maps a pseudo-digraph to a bidirected pseudo-digraph by appropriately extending the set of edges with missing reversed edges. A vertex v is *reachable* from a vertex u in a (bi)directed graph G if there exists a directed *walk* from u to v in G, i.e., a sequence $(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)$ of edges with $u_1 = u$, $v_k = v$ and $v_i = u_{i+1}$ for $1 \leq i < k$. An alternative and equivalent formalization of reachability between vertices u and v in G is via sequences of vertices $v_1, v_2, \ldots, v_k$ where $v_1 = u$, $v_k = v$ and (v_i, v_{i+1}) is an edge of G for $1 \leq i < k$. An undirected graph is *connected* if for any two vertices u and v of the graph u is reachable from v.

Our high-level proof rests on the Isabelle locale *connected-components-locale* (Listing 4) that describes the assumptions of our theorem. We fix G to be a pseudo-digraph where α is an abstraction of the type of vertices and β is an abstraction of the type of edges. We furthermore fix a representation of the spanning tree witnessing connectivity (cf. Section 4.1.2) with functions *parent-edge* and *num* and vertex r as the root. Based on these assumptions we prove that G is connected. We show first that every vertex v in the graph is reachable from the root r by induction on *num* v, i.e., the length of the walk from r to v in the spanning tree. The base case follows directly from our assumptions. For the inductive step, we can assume a walk from r to the parent of a vertex v. Using the assumptions, this walk can be extended to a walk from r to v since there is an edge between v and its parent. Now, since G is bidirected, we can establish that there is a walk between any two vertices of G by combining the walks that connect them with the root r.

The final part of the formal proof, i.e., linking the high-level proofs with the properties exported from VCC to Isabelle, is mostly straightforward. Proving that



vertex	s	t	u	v	w
parent-edge	$\perp$	0	1	3	$\perp$
num	0	1	1	2	∞
dist	0	1	1	1	∞

(a) A directed graph G (b) A shortest-path tree of G (c) Tree representation

Fig. 4: A directed graph $G = (V, E)$ with the edges labeled $i/\mathbf{k}$, where i is a unique edge index and where $\mathbf{k}$ is the cost of that edge, and a shortest-path tree of G rooted at start vertex $s \in V$. The tree is encoded by *parent-edge*, *num* and *dist* according to the table in (c). Observe that vertex w is not reachable from s and that the cycle $t \to v \to t$ has cost zero.

the precondition and the witness predicate (cf. Section 4.1.2) match the assumptions specified in the locale *connected-components-locale* involves no reasoning beyond syntactical rewriting. For instantiating the assumptions we provide lifting functions that abstract from the concrete representation of graphs and spanning trees stemming from our VCC specification to the high-level representation used by the Isabelle graph library. It follows that a lifted, high-level graph is connected. Establishing the checker postcondition, i.e., connectivity of unlifted graphs, hence requires to show that any high-level path witnessing reachability between two vertices has an unlifted path. This is almost direct since our representation of paths in the VCC formalization (cf. Section 4.1.2) is close to the path representation of the Isabelle graph library.

4.2 Case Study: Shortest Paths in Graphs

The single-source shortest-paths problem in directed graphs with nonnegative edge weights can be solved, for instance, by Dijkstra's algorithm [24, Sections 6.6 and 7.5]. Instead of verifying this algorithm, we request that it returns, along with the computed shortest distance for each vertex of a graph, the corresponding shortest path as witness. That is, we instantiate our general approach as follows:

input x	$=$	a directed graph $G = (V, E)$, a function $cost : E \to \mathbb{N}$ for edge weights, a vertex s
output y	$=$	a mapping $dist : V \to (\mathbb{N} \cup \infty)$
witness w	$=$	a tree rooted at s
$\varphi(x)$	$=$	G is wellformed and $s \in V$
$\mathcal{W}(x, y, w)$	$=$	w is a shortest-path tree, i.e., for each v reachable from s, the tree path from s to v has length $dist(v)$
$\psi(x, y)$	$=$	for each $v \in V$, $dist(v)$ is the cost of a shortest path from s to v (∞, if there is no path from s to v).

Figure 4 shows a directed graph and a shortest-path tree rooted at s. We encode a shortest-path tree by functions *parent-edge*, *dist*, and *num*. For each v reachable from s, $dist(v)$ is the shortest-path distance from s to v and $num(v)$ is the depth of v in the shortest-path tree. For vertices v that are not reachable from s, $dist(v) = num(v) = \infty$. For reachable vertices v different from s the edge $parent\text{-}edge(v)$ is the last edge on a shortest path from s to v. This witness is somewhat verbose. As we will see in Section 4.2.3 we could do without the *parent-edge* function. If all edge costs are positive, no witness would be required beyond the *dist* function as output. If one allows also cost zero for edges as we do, the depth function *num* is indispensable; see [23, Section 2.4].

We present the implementation of a checker in Section 4.2.1, detail the formalization of the witness predicate and the verification of the checker in Section 4.2.2 and prove the witness property in Section 4.2.3. The relevant files in the companion website are check_shortest_path.c (C-code and checker correctness), check_shortest_path.thy (Isabelle/HOL representation of the VCC checker specification and witness property), ShortestPath.thy (abstract verification of the witness property in a locale), and ShortestPath_Link.thy (instantiation of the locale with the Isabelle/HOL representation of the VCC specification).

4.2.1 Shortest-Paths Checker

We adopt the data structures of the previous case study (Section 4.1.1) with the exception that the num array stores elements of type **int** instead of Nat. This is because vertices may now also be unreachable, i.e., there is no path from the source vertex to them, and we encode this by requiring that num takes a negative value for such vertices. We represent distances from the source vertex to any other vertex by an array dist with elements of type **int**. Any negative value encodes ∞. Finally, the edge weights are modeled by an array cost that gives for every edge a value of type ushort (an abbreviation for **unsigned short**).

Based on these types, we implement the shortest-path checker as a function that succeeds when all of the four functions given in Listing 5 succeed. That is, we check that the source vertex s is indeed the starting point (in check_start_val), that the dist and num arrays are consistent with respect to unreachable vertices, i.e., either both are finite or both are infinite (in check_no_path), that the triangle inequality property (see Section 4.2.3) is fulfilled (in check_trian), and that the parent edge of every vertex v defines its distance value (in check_just).

There is a subtle point in the checker code. We want to establish the triangle inequality ($dist(u) + cost(u, v) \geq dist(v)$ for all edges (u, v)) and the distance justification ($dist(u) + cost(u, v) = dist(v)$ if (u, v) is the parent edge of v) over the extended natural numbers $\mathbb{N} \cup \{\infty\}$. However, C knows only finite precision arithmetic. We solve the case of infinite distances by appropriate case distinctions. We solve the case of potential overflow in finite precision arithmetic as follows: Distances are of type **int**, i.e., from the set $\{-2^{31}, \ldots, 2^{31} - 1\}$ on a 32-bit platform, and edge costs are of type ushort, i.e., between 0 and $2^{16} - 1$, and hence contained in the set of nonnegative values of type **int**. In arithmetic expressions, we cast all nonnegative values to **unsigned** with range $0 \ldots 2^{32} - 1$. This guarantees that finite precision arithmetic is exact and allows VCC to conclude equalities and inequalities between natural numbers.

If parent_edge is not part of the witness, check_just needs to be rewritten. When considering a node v it has to iterate over all edges into v to find the edge that defines dist[v]. An efficient implementation of this iteration requires to provide each vertex with the list of edges into it.

4.2.2 Checker Correctness

We now define our abstract specification for the shortest-path checker. We use the same data structures as in the previous case study (see Section 4.1.2) with the exception that the num mapping now takes vertices to extended naturals, i.e., to the set $\mathbb{N} \cup \{\infty\}$, represented by the type Enat that provides an explicit value for infinity:

```
_(datatype \Enat
{
   case \enat_inf();
   case \enat_val(\natural n);
})
```

We define functions \is_enat_inf to check whether an extended natural is infinity and \enat_val_of to convert an extended natural which is not infinity into a natural number. For better readability, we will sometimes write $\mathsf{a} =_e \infty$ instead of \is_enat_inf(a). Moreover, we provide the predicates \enat_eq (abbreviated by $=_e$) and \enat_le (abbreviated by $\leq_e$) to decide equality and less-or-equal of two extended naturals as well as a function \enat_add (also written as $+_e$) for the sum of an extended natural and a natural number:

```
_(def \bool \enat_eq(\Enat e1, \Enat e2)
{
   return
      (e1 =_e ∞ ∧ e2 =_e ∞) ∨
      (e1 ≠_e ∞ ∧ e2 ≠_e ∞ ∧ \enat_val_of(e1) = \enat_val_of(e2));
})

_(def \bool \enat_le(\Enat e1, \Enat e2)
{
   return e2 =_e ∞ ∨ (e1 ≠_e ∞ ∧ e2 ≠_e ∞ ∧ \enat_val_of(e1) ≤ \enat_val_of(e2));
})

_(def \Enat \enat_add(\Enat e, \natural n)
{
   return (e =_e ∞) ? \enat_inf() : \enat_val(\enat_val_of(e) + n);
})
```

The type of extended natural numbers is also used for the abstract representation of the dist array. Again, as in the previous case study, concrete types and abstract types are sufficiently similar such that abstraction functions to relate one to the other are straightforward to define. We omit them here.

The preconditions of this case study are that G is a wellformed graph and that the source vertex s is a vertex of G, i.e., that s < G.num_verts holds. We formalize the witness predicate as a conjunction of four properties, one for each of the four checker functions in Listing 5.

check_start_val: Vertex s is indeed the starting point:

$$\mathsf{dist[s]} =_e \mathsf{\backslash enat\_val(0)}$$

```
bool check_start_val(Vertex s, int* dist, int* num)
{
    return dist[s] == 0;
}

bool check_no_path(Graph* G, int* dist, int* num)
{
    Vertex v;

    for (v = 0; v < G->n; v++)
    {
        if (INF(dist[v]) != INF(num[v])) return FALSE;
    }
    return TRUE;
}

int check_trian(Graph* G, ushort* cost, int* dist)
{
    Edge_Id e; Vertex source, target;

    for (e = 0; e < G->m; e++)
    {
        source = G->es[e].s;
        target = G->es[e].t;

        if (INF(dist[source])) continue;
        if (INF(dist[target])) return FALSE;
        if (VAL(dist[target]) > VAL(dist[source]) + cost[e]) return FALSE;
    }
    return TRUE;
}

bool check_just(Graph* G, Vertex s, ushort* cost, int* dist, int* parent_edge, int* num)
{
    Vertex v, source; Edge_Id e;

    for (v = 0; v < G->n; v++)
    {
        if (v == s || INF(num[v])) continue;
        if (parent_edge[v] < 0 || ((Edge_Id)parent_edge[v]) >= G->m) return FALSE;

        e = (Edge_Id)parent_edge[v];
        source = G->es[e].s;

        if (G->es[e].t != v) return FALSE;
        if (INF(dist[source]) || VAL(dist[v]) != VAL(dist[source]) + cost[e]) return FALSE;
        if (INF(num[source]) || VAL(num[v]) != VAL(num[source]) + 1) return FALSE;
    }
    return TRUE;
}
```

Listing 5: Functions comprising the shortest-path checker. We use the predicate INF(x) to abbreviate $x < 0$, and VAL(x) stands for the type cast (Nat)x; Nat is the C type **unsigned** as defined in Listing 1.

check_no_path: The num mapping and the dist mapping are consistent with respect to unreachable vertices, i.e., either both are finite or both are infinite:

$\forall$ \Vertex v; v $<$ G.num_verts $\longrightarrow$ (dist[v] $=_e \infty \longleftrightarrow$ num[v] $=_e \infty$)

check_trian: The triangle inequality holds for all edges of the graph:

```
∀ \Edge_Id i; i < G.num_edges ⟶
    dist[G.edge[i].trg] ≤e dist[G.edge[i].src] +e cost[i]
```

check_just: The parent edges encode a tree rooted at s and define the distance values of reachable vertices:

```
∀ \Vertex v;
    v < G.num_verts ∧ v ≠ s ∧ num[v] ≠e ∞ ⟶
    \is_some(parent_edge[v]) ∧ \the(parent_edge[v]) < G.num_edges ∧
    v = G.edge[\the(parent_edge[v])].trg ∧
    dist[v] =e dist[G.edge[\the(parent_edge[v])].src] +e cost[\the(parent_edge[v])] ∧
    num[v] =e num[G.edge[\the(parent_edge[v])].src] +e 1
```

We verified that each of these four properties holds if and only if the corresponding checker function succeeds. The three functions check_no_path, check_trian and check_just need additional annotations before VCC can verify their correctness. The loops in these functions have to be annotated with loop invariants which are, just as in the previous case study (see Section 4.1.2), only simple variants of the postconditions above. Also, as for the connected-components checker, we need to state explicitly properties that are violated before every **return** FALSE statement. Such properties are reformulations of concrete properties on the abstract level. In addition, both check_trian and check_just require that the graph under consideration is wellformed, and check_just furthermore requires that num and dist are consistent (the postcondition of check_no_path). We add these requirements as preconditions to the checker functions.

For being able to express the postcondition of the shortest-path checker, we define sequences of edges as a recursive datatype:

```
_(datatype \Path
{
    case none();
    case path(\Edge_Id i, \Path p);
})
```

Only particular instances of this datatype are indeed paths in the given graph G. To qualify valid paths, we proceed in two steps. We first define a predicate that expresses the conditions under which a sequence of edges constitutes a walk in graph G from vertex u to vertex v (see Listing 6). Second, we define a predicate to describe when the set of vertices of an edge sequence is distinct (see Listing 7). A path from vertex u to vertex v in G is a walk p from u to v with distinct vertices. We define this as a predicate \is_path(G, p, u, v).

With a recursive function \path_cost that computes for a given path its length using the cost mapping, we can finally state the postcondition of the shortest path checker:

```
(∀ \Vertex v; v < G.num_verts ⟶
    ¬\is_enat_inf(dist[v]) ⟷ (∃ \Path p; \is_path(G, p, s, v))) ∧
(∀ \Vertex v; v < G.num_verts ∧ ¬\is_enat_inf(dist[v]) ⟶
    (∀ \Path p; \is_path(G, p, s, v) ⟶ \enat_val_of(dist[v]) ≤ \path_cost(cost, p)) ∧
    (∃ \Path p; \is_path(G, p, s, v) ∧ \enat_val_of(dist[v]) = \path_cost(cost, p)))
```

```
_(def \bool \is_walk(\Graph G, \Path p, \Vertex u, \Vertex v)
{
  switch (p)
  {
    case none(): return u = v;
    case path(i, q):
      return i < G.num_edges ∧ u = G.edge[i].src ∧ \is_walk(G, q, G.edge[i].trg, v);
  }
})
```

Listing 6: A walk from vertex u to vertex v is a finite sequence of connected edges of graph G where the source vertex of the first edge is u and the target vertex of the last edge is v.

```
_(def \bool \occurs(\Graph G, \Vertex u, \Vertex v, \Path p)
  _(decreases \size(p))
{
  switch (p)
  {
    case none(): return u = v;
    case path(i, q): return u = G.edge[i].src ∨ \occurs(G, u, G.edge[i].trg, q);
  }
})

_(def \bool \distinct_verts(\Graph G, \Path p)
{
  switch (p)
  {
    case none(): return \true;
    case path(i, q): return ¬\occurs(G, G.edge[i].src, G.edge[i].trg, q) ∧ \distinct_verts(G, q);
  }
})
```

Listing 7: Predicate \distinct_verts(G, p) holds if the set of vertices connected by path p is distinct. Predicate \occurs(G, u, v, p) is true if and only if u is either equal to v or equal to any vertex touched by path p.

We formally prove the validity of this property, under the assumption of the precondition and the witness predicate, in Isabelle in the next section.

4.2.3 Proof of the Witness Property for the Shortest-Path Checker

We now prove in Isabelle the witness property, i.e., if G is wellformed (the precondition of this case study) and if the witness predicate holds then for all vertices $v \in V$, $dist(v)$ is indeed the shortest-path distance from s to v.

Listing 8 shows our Isabelle locales containing the assumptions we make. We separate the assumptions into three locales to avoid the use of unneeded assumptions when proving intermediate lemmas. This makes the intermediate lemmas more general and hence usable in other contexts. For example, we used some of the lemmas also for the verification of a checker for the shortest-path problem with general edge weights (not only nonnegative edge weights as in this

```
locale basic-sp = pseudo-digraph +
  fixes dist :: α ⇒ ereal
  fixes c :: β ⇒ real
  fixes s :: α
  assumes general_source_val: dist s ≤ 0
  assumes trian: ⋀e. e ∈ edges G ⟹ dist (target G e) ≤ dist (start G e) + c e

locale basic-just-sp = basic-sp +
  fixes num :: α ⇒ enat
  assumes just:
   ⋀v. v ∈ verts G ⟹ v ≠ s ⟹ num v ≠ ∞ ⟹
      ∃ e ∈ edges G.
        v = target G e ∧
        dist v = dist (start G e) + c e  ∧
        num v = num (start G e) + (enat 1)

locale shortest-path-pos-cost = basic-just-sp +
  assumes s_in_G: s ∈ verts G
  assumes start_val: dist s = 0
  assumes no_path: ⋀v. v ∈ verts G ⟹ dist v = ∞ ⟷ num v = ∞
  assumes pos_cost: ⋀e. e ∈ edges G ⟹ 0 ≤ c e
```

Listing 8: Locales for the shortest-paths proof in Isabelle

case study) [33]. The first locale *basic-sp* subsumes the locale *pseudo-digraph* that is mentioned in Section 4.1.3. Moreover, we assume we are given the function $dist : V \to (\mathbb{R} \cup \{\infty, -\infty\})$, an edge cost function $c : E \to \mathbb{R}$, and a start vertex s.

We lift c to a function *cost* that takes a walk as input and returns the sum of the costs of its edges. Using the *basic-sp* locale we prove that for every vertex v and walk p from s to v, $dist\ v \leq cost\ p$. We prove this by induction on the length of p. If the length of p is 0 then its cost is 0 and $s = v$. Using the general_source_val assumption we know $dist\ s \leq 0$, and hence $dist\ s \leq cost\ p$. For the inductive case, let p be a path $p\prime$ with an appended edge e, denoted as $p\prime \bullet e$ where $e = (u, v)$. Then by the induction hypothesis we know $dist\ u \leq cost\ p\prime$ and by the trian assumption we know $dist\ v \leq dist\ u + c\ e$, and thus $dist\ v \leq cost\ p\prime + c\ e$, from which we conclude $dist\ v \leq cost\ p$.

Let μ be a function that takes two vertices s and v and a cost function c on edges and returns the cost of a shortest path from s to v in G using the cost function c. Using the locale *basic-sp* and the fact that for any vertex v and walk p from s to v, $dist\ v \leq cost\ p$ we can also deduce that $dist\ v \leq \mu\ v$ for every vertex $v \in V$. We call this lemma *dist-le-*μ.

We use the locale *basic-just-sp* to prove that $dist\ v \geq \mu\ v$ for every vertex $v \in V$ under some extra assumptions (see Listing 9) and call this lemma *dist-ge-*μ. Later we show that these extra assumptions hold in the locale *shortest-path-pos-cost*. The proof of *dist-ge-*μ proceeds by induction over $num\ v$. If $num\ v = 0$ then $v = s$ (since $num\ v \neq 0$ for every $v \neq s$) and hence, since we assumed $dist\ s = 0$ and $\mu\ s = 0$, we have $dist\ s \geq \mu\ s$. For the inductive step, by the assumption named just we know that there is an edge $e = (u, v)$ in E such that $dist\ v = dist\ u + c\ e$ and $num\ v = num\ u + 1$. By the induction hypothesis and the lemma *dist-le-*μ we know $dist\ u = \mu\ u$ and hence $dist\ v = \mu\ u + c\ e$. If $dist\ v = \infty$ then $dist\ v \geq \mu\ v$. Otherwise we have $dist\ v \neq \infty$ and, since we assumed $dist\ v \neq -\infty$, we also have $\mu\ u \neq \infty$

lemma (**in** *basic-just-sp*) *dist-ge-μ*:
  fixes $v :: \alpha$
  assumes $v \in \mathit{verts}\ G$
  assumes $\mathit{num}\ v \neq \infty$
  assumes $\mathit{dist}\ v \neq -\infty$
  assumes $\mu\ c\ s\ s = \mathit{ereal}\ 0$
  assumes $\mathit{dist}\ s = 0$
  assumes $\bigwedge u.\ u \in \mathit{verts}\ G \Longrightarrow u \neq s \Longrightarrow \mathit{num}\ u \neq \infty \Longrightarrow \mathit{num}\ u \neq \mathit{enat}\ 0$
  shows $\mathit{dist}\ v \geq \mu\ c\ s\ v$

Listing 9: The central lemma of the shortest-paths proof in Isabelle

and $\mu\ u \neq -\infty$. Hence, there must be a walk p from s to u with $\mu\ u = \mathit{cost}\ p$. By definition of μ we know $\mu\ v \leq \mathit{cost}\ (p \bullet e)$ and by definition of *cost* we thus have $\mu\ v \leq \mu\ u + c\ e$. Therefore, $\mathit{dist}\ v \geq \mu\ v$.

Now we show that the extra assumptions that we made while proving lemma *dist-ge-μ*, i.e., all assumption except for the first one (Listing 9), hold under the *shortest-path-pos-cost* locale.

The assumption $\mathit{num}\ v \neq \infty$ is superfluous, for if $\mathit{num}\ v$ is equal to ∞, so is $\mathit{dist}\ v$ using the no_path assumption, and then the inequality $\mathit{dist}\ v \geq \mu\ v$ holds as well. The assumption $\mu\ s\ s\ c = 0$ holds using the pos_cost assumption, since if all edges have nonnegative cost then so do all walks and thus the empty walk from s to itself is the least cost walk to s; recall that the empty walk has cost zero. The assumption $\mathit{dist}\ s = 0$ holds directly by the start_val assumption. The last assumption of *dist-ge-μ*, i.e., for every vertex $u \in V$ with $u \neq s$ we have that $\mathit{num}\ u \neq \infty$ implies $\mathit{num}\ u \neq 0$, holds using the just assumption. We refer to this assumption of *dist-ge-μ* with u instantiated by v as *num-not0* lemma. Finally, we show how to fulfill the assumption $\mathit{dist}\ v \neq -\infty$. If $\mathit{num}\ v = \infty$ then $\mathit{dist}\ v = \infty$ by the no_path assumption, and we are done. For $\mathit{num}\ v \neq \infty$ we prove by induction on $\mathit{num}\ v$ that $\mathit{dist}\ v \neq -\infty$. If $\mathit{num}\ v = 0$ we know that $v = s$ using the *num-not0* lemma and therefore $\mathit{dist}\ v \neq -\infty$ using the start_val assumption. For the inductive step, if $v = s$ then $\mathit{dist}\ v = 0$ by the start_val assumption, and we are done. Otherwise, if $v \neq s$, we know by the just assumption that there is an edge $e = (u, v)$ in E such that $\mathit{dist}\ v = \mathit{dist}\ u + c\ e$ and $\mathit{num}\ v = \mathit{num}\ u + 1$. By $\mathit{num}\ v \neq \infty$ we have $\mathit{num}\ u \neq \infty$ and hence by the induction hypothesis $\mathit{dist}\ u \neq -\infty$. Consequently $\mathit{dist}\ v \neq -\infty$. This concludes the proofs of the extra assumptions made in the lemma *dist-ge-μ* under the locale *shortest-path-pos-cost*. Thus, we know that under this locale $\mathit{dist}\ v = \mu\ v$ for all $v \in V$.

Linking this Isabelle proof with the specification exported from VCC is a matter of translating from one representation to another. We chose to define paths and their costs in VCC (see Section 4.2.2) intentionally similar to the way they are defined in the Isabelle graph library to ease our translation proofs. Since there are several more concepts to relate than in the previous checker (see Section 4.1.3), our proofs for the shortest-path checker are more tedious. Nevertheless, no complex reasoning is required. We establish that the assumptions of the *shortest-path-pos-cost* locale are implied by the checker precondition and witness predicate, and we prove that our final theorem proved in that locale implies the checker postcondition.

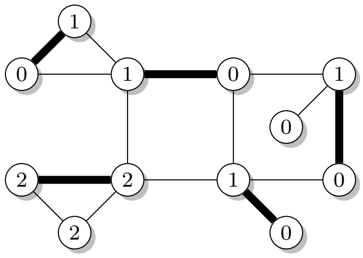


Fig. 5: The vertex labels certify that the indicated matching is of maximum cardinality: All edges of the graph have either both endpoints labeled as 2 or at least one endpoint labeled as 1. Any matching can hence use at most one edge with both endpoints labeled 2 and at most four edges that have an endpoint labeled 1. Therefore, no matching has more than five edges. The matching shown consists of five edges (in bold).

4.3 Case Study: Maximum Cardinality Matching in Graphs

A *matching* in a graph G is a subset M of the edges of G such that no two share an endpoint. A matching has maximum cardinality if its cardinality is at least as large as that of any other matching. Figure 5 shows a graph, a maximum cardinality matching, and a witness of this fact. An *odd-set cover* L of a graph G is a labeling of the vertices of G with integers such that every edge of G is either incident to a vertex labeled 1 or connects two vertices labeled with the same number i and $i \geq 2$.

Theorem 2 (Edmonds [16]) *Let M be a matching in a graph G and let L be an odd-set cover of G. For any $i \geq 0$, let n_i be the number of vertices labeled i. If*

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor \tag{2}$$

then M is a maximum cardinality matching.

Proof Let N be any matching in G. For $i \geq 2$, let N_i be the edges in N that connect two vertices labeled i and let N_1 be the remaining edges in N. Then, by the definition of odd-set cover, every edge in N_1 is incident to a vertex labeled 1. Since edges in a matching do not share endpoints, we have

$$|N_1| \leq n_1 \quad \text{and} \quad |N_i| \leq \lfloor n_i/2 \rfloor \quad \text{for } i \geq 2.$$

Thus $|N| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor = |M|$. □

It can be shown (but this is nontrivial) that for every maximum cardinality matching M there is an odd-set cover L satisfying Equality (2). The cover uses nonnegative vertex labels in the range 0 to $|V| - 1$ and all n_i's with $i \geq 2$ are odd. The *certifying algorithm for maximum cardinality matching* in LEDA returns a matching M and an odd-set cover L such that (2) holds. The relationship to Section 2 is as follows:

$$
\begin{array}{rcl}
\text{input } x & = & \text{an undirected graph } G \\
\text{output } y & = & \text{a set of edges } M \\
\text{witness } w & = & \text{a vertex labeling } L \\
\varphi(x) & = & G \text{ and } M \text{ are wellformed and have no self loops} \\
\mathcal{W}(x,y,w) & = & M \text{ is a matching in } G,\ L \text{ is an odd-set cover for } G, \text{ and} \\
 & & \text{Equation (2) holds} \\
\psi(x,y) & = & M \text{ is a maximum cardinality matching in } G.
\end{array}
$$

Theorem 2 is the witness property. We give a formal proof for it in Section 4.3.3. Writing a correct program which checks whether a set of edges is a matching and a vertex labeling is an odd-set cover which together satisfy Equation 2 is easy, see Section 4.3.1. In Section 4.3.2 we describe the verification of such a checker. The relevant files in the companion website are check_matching.c (C-code and checker correctness), check_matching.thy (Isabelle/HOL representation of the VCC checker specification and witness property), Matching.thy (abstract verification of the witness property in a locale), and Matching_Link.thy (instantiation of the locale with the Isabelle/HOL representation of the VCC specification).

This case study is a modified version of the one we present in [1]. The proof of the witness property is very similar to the one published in [32] except that it uses the Isabelle graph library which was developed meanwhile.

4.3.1 Maximum-Cardinality-Matching Checker

We build the checker using the graph data structure as in the previous case studies (see Listing 1). We assume that graphs are wellformed and have neither self loops nor duplicate edges. We treat the edges of a graph as undirected edges. Matchings are also represented by graphs. We require an additional witness in the form of an array `f` mapping edge identifiers of the matching to edge identifiers of the input graph. For instance, if a graph consists of three edges (identified as 0, 1 and 2) and the computed matching consists of the third edge (i.e., 2), then `f` would be an array with a single element 2 indicating how the only edge of the matching corresponds to the edges of the input graph. Finally, the vertex labeling is represented by an array `osc` which is indexed by vertices and which stores elements of type `Nat`. The checker function requires an auxiliary array `check` that can store as many elements of type `Nat` as there are vertices in the input graph, but at least two. We expect that this array is allocated elsewhere and given as extra input to the checker.

In addition to the checker function, there are four helper functions, see Listing 10. The checker succeeds if the first three of them succeed and if the fourth function returns a value that is equal to the number of edges of the matching `M`. In short, the helper functions perform the following tasks. The function `check_subset` checks whether `M` is a subgraph of `G` with respect to the mapping `f`. In `check_matching`, we check that `M` is indeed a matching, i.e., that it contains no two edges which are incident. With `check_osc` the vertex labeling is checked to be an odd-set cover and that vertex labels are in the range $\{0, \ldots, \mathsf{G}{-}{>}\mathsf{n} - 1\}$. Finally, the function `weight` computes the sum on the right-hand side of Equation (2). This computation is optimized by first searching for the greatest vertex label, which can be considerably smaller than the maximal $\mathsf{G}{-}{>}\mathsf{n} - 1$, and then summing up partial sums only until this greatest label. The main checker function passes the auxiliary array `check` to `check_matching` as `degree_in_M` argument and to `weight` as `count` argument.

```
bool check_subset(Graph* G, Graph* M, Nat* f)
{
    Edge_Id e;

    for (e = 0; e < M->m; e++)
    {
        if (f[e] >= G->m) return FALSE;
        if (M->es[e].s == G->es[f[e]].s && M->es[e].t == G->es[f[e]].t) continue;
        if (M->es[e].s == G->es[f[e]].t && M->es[e].t == G->es[f[e]].s) continue;
        return FALSE;
    }
    return TRUE;
}

bool check_matching(Graph* M, Nat* degree_in_M)
{
    Vertex v; Edge_Id e;

    for (v = 0; v < M->n; v++) degree_in_M[v] = 0;
    for (e = 0; e < M->m; e++)
    {
        if (degree_in_M[M->es[e].s] == 1 || degree_in_M[M->es[e].t] == 1) return FALSE;
        degree_in_M[M->es[e].s] = 1;
        degree_in_M[M->es[e].t] = 1;
    }
    return TRUE;
}

bool check_osc(Graph* G, Nat* osc)
{
    Edge_Id e; Vertex v, w;

    for (v = 0; v < G->n; v++) if (osc[v] >= G->n) return FALSE;
    for (e = 0; e < G->m; e++)
    {
        v = G->es[e].s;
        w = G->es[e].t;
        if (osc[v] == 1 || osc[w] == 1 || (osc[v] == osc[w] && osc[v] ≥ 2)) continue;
        return FALSE;
    }
    return TRUE;
}

Nat weight(Graph* G, Nat* osc, Nat* count)
{
    Vertex v; Nat c, s, max = 1, r = (G->n > 2) ? G->n : 2;

    for (c = 0; c < r; c++) count[c] = 0;
    for (v = 0; v < G->n; v++)
    {
        count[osc[v]] = count[osc[v]] + 1;
        if (osc[v] > max) max = osc[v];
    }
    s = count[1];
    for (c = 2; c < max + 1; c++) s += count[c] / 2;
    return s;
}
```

Listing 10: Helper functions of the maximum-cardinality-matching checker

4.3.2 Checker Correctness

We build on the abstract graph data structure of Listing 3. We require that graphs are are wellformed and contain neither self loops:

$\forall$ \Edge_Id i; i < G.num_edges $\longrightarrow$ G.edge[i].src $\neq$ G.edge[i].trg;

nor duplicate edges:

$\forall$ \Edge_Id i1, i2; i1 < G.num_edges $\wedge$ i2 < G.num_edges $\wedge$ i1 $\neq$ i2 $\longrightarrow$
G.edge[i1].src $\neq$ G.edge[i2].src $\vee$ G.edge[i1].trg $\neq$ G.edge[i2].trg;

An abstract vertex labeling L is a mapping from vertices to natural numbers. The mapping f from edge identifiers to edge identifiers has a straightforward representation as abstract mapping. We omit here, as in the previous case studies, the description of abstraction functions from concrete to abstract values.

The witness predicate is a conjunction of four predicates, each related to one of the helper functions in Listing 10.

check_subset: M must be a subgraph of G w.r.t. the edge mapping f, i.e., every edge of M must also be an edge of G modulo symmetry of edges:

$\forall$ \Edge_Id i; i < M.num_edges $\longrightarrow$
f[i] < G.num_edges $\wedge$
(M.edge[i].src = G.edge[f[i]].src $\wedge$ M.edge[i].trg = G.edge[f[i]].trg $\vee$
M.edge[i].src = G.edge[f[i]].trg $\wedge$ M.edge[i].trg = G.edge[f[i]].src)

check_matching: M must be a matching, i.e., no two edges of M have a vertex in common:

$\forall$ \Edge_Id i1, i2;
i1 < M.num_edges $\wedge$ i2 < M.num_edges $\wedge$ i1 $\neq$ i2 $\longrightarrow$
M.edge[i1].src $\neq$ M.edge[i2].src $\wedge$ M.edge[i1].src $\neq$ M.edge[i2].trg $\wedge$
M.edge[i1].trg $\neq$ M.edge[i2].src $\wedge$ M.edge[i1].trg $\neq$ M.edge[i2].trg

check_osc: L must be an odd-set cover of G, i.e., for every edge of G one of the edge's vertices is labeled 1 or both vertices are labeled by the same number which is greater or equal 2:

$\forall$ \Edge_Id i; i < G.num_edges $\longrightarrow$
L[G.edge[i].src] = 1 $\vee$
L[G.edge[i].trg] = 1 $\vee$
L[G.edge[i].src] = L[G.edge[i].trg] $\wedge$ L[G.edge[i].src] $\geq$ 2

weight: Equation (2) must hold. We define it stepwise. The number of vertices labeled with c is defined recursively:

```
_(def \natural \label_count(\Label L, \natural c, \natural i)
{
   return (i = 0) ? 0 : ((L[i − 1] = c) ? 1 : 0) + \label_count(L, c, i − 1);
})
```

We have n_{c} = \label_count(L, c, G.num_verts) for a vertex label c. The sum of these numbers for labels greater than 1 is again defined recursively:

```
_(def \natural \rec_weight(\Label L, \natural n, \natural i)
{
   return (i < 2) ? 0 : \label_count(L, i, n) / 2 + \rec_weight(L, n, i − 1);
})
```

We have $\sum_{i\geq 2}\lfloor n_i/2\rfloor$ = \rec_weight(L, G.num_verts, m) where m is the greatest label that is assigned to any vertex by L. The complete sum is then:

```
_(def \natural \full_weight(\Label L, \natural n, \natural i)
{
  return \label_count(L, 1, n) + \rec_weight(L, n, i);
})
```

That is, we have $n_1 + \sum_{i\geq 2}\lfloor n_i/2\rfloor$ = \full_weight(L, G.num_verts, m) with the same m as before. Finally, the predicate capturing Equation (2) is as follows:

```
M.num_edges = \full_weight(L, G.num_verts, m) ∧
∀ \Vertex v; v < G.num_verts ⟶ L[v] ≤ m
```

Verifying the correctness of the checker (Section 4.3.1) follows the lines of the earlier case studies for the first three predicates above. We only have to provide the right loop invariants, and simple variations of the predicates to be proved are sufficient. In check_matching, we need additional loop invariants. Along with the first loop we accumulate the knowledge about the initialization of the degree_in_M array, i.e., the first positions of the array have already been set 0:

```
∀ Nat u; u < v ⟶ degree_in_M[u] = 0
```

Moreover, on the second loop we need three additional loop invariants. One invariant maintains that values stored in degree_in_M are in range:

```
∀ Nat v; v < M−>n ⟶ degree_in_M[v] ≤ 1
```

Another invariant specifies the knowledge about vertices for which degree_in_M is still 0, i.e., that those vertices cannot be part of any already checked edge:

```
∀ Nat v; v < M−>n ∧ degree_in_M[v] = 0 ⟶
  ∀ Nat e1; e1 < e ⟶ M−>es[e1].s ≠ v ∧ M−>es[e1].t ≠ v
```

And finally, vertices for which degree_in_M has already been set to 1 are mapped by a ghost mapping E to their adjacent edge in the matching M:

```
∀ Vertex v; v < M−>n ∧ degree_in_M[v] = 1 ⟶
  E[v] < e ∧ (M−>es[E[v]].s = v ∨ M−>es[E[v]].t = v)
```

This invariant is required to prove completeness. We maintain this invariant by updating the ghost mapping E in the loop body accordingly.

We need that G is wellformed to establish correctness of check_osc and for the verification of check_matching, in the latter case together with the fact that G has no self loops.

Proving correct the weight function is the most intricate part of the checker verification. There are two properties to show, functional correctness and absence of overflows. The former is to show that the function computes the n_i and the overall sum of Equation (2) correctly as specified in the above fourth conjunct of the witness predicate. The latter is to show that the additions in both the second and third loop do not overflow. Surprisingly, the absence of overflows is much harder to establish than functional correctness. We concentrate on functional correctness first.

The definition of the \label_count function is such that we can maintain the following property in the second loop:

```
∀ Nat j; j < r ⟶ count[j] = \label_count(L, j, v)
```

With this property, the following loop invariant on the third loop can be maintained:

```
s = \full_weight(L, G−>n, c − 1))
```

Together with a further loop invariant for the second loop to guarantee that max is the greatest label seen so far, we can conclude that the weight function is functionally correct.

The addition in the second loop can never overflow because in each loop iteration, the loop variable is an upper limit on the value of count[i] array for each label i. Concerning the addition in the third loop, we observe that in each loop iteration the value of s is bounded by the number of vertices in G. To establish this property, we build up a ghost map sum in the second loop in such a way that in every iteration of that loop this map fulfills the following invariant:

```
sum[1] = count[1] ∧
(∀ Nat j; 1 < j ∧ j < r ⟶ sum[j] = sum[j − 1] + count[j]) ∧
(∀ Nat j; 1 < j ∧ j < r ⟶ sum[j] ≤ v)
```

Maintaining this invariant requires to update the sum map during each iteration of the second loop. We do so in a nested ghost loop where we propagate the increment that happened on the count array to every possibly affected element sum[j].

The postcondition of the checker expresses that the cardinality of any matching of G cannot be smaller than the cardinality of M:

```
∀ \Graph M2; \Edge_Map I2; \is_subset(G, M2, I2) ∧ \is_matching(M2) ⟶
   M2.num_edges ≤ M.num_edges;
```

We give a formal proof that the checker preconditions and the witness predicate imply this property in the following section.

4.3.3 Proof of the Witness Property for the Maximum-Cardinality-Matching Checker

We explain the Isabelle proof for the witness property, i.e., Theorem 2. See Listing 11 for an excerpt of our formal Isabelle proof development that can be found in file Matching.thy. The formal proof follows the scheme of the nonformal proof and is split into two main parts.

For $i \geq 2$, let M_i be the edges in M that connect two vertices labeled i and let M_1 be the remaining edges in M. We use the definition of an odd-set cover to prove that $M \subseteq \bigcup_{i\geq 1} M_i$ and thus $|M| \leq \sum_{i\geq 1} |M_i|$. Let V_i be the vertices labeled i and let $n_i = |V_i|$. We formally prove: $|M_1| \leq n_1$ and $|M_i| \leq \lfloor n_i/2 \rfloor$.

In order to prove $|M_1| \leq n_1$, we exhibit an injective function from M_1 to V_1. We first prove, using the definition of an odd-set cover, that every edge $e \in M_1$ has at least one endpoint in V_1. This gives rise to a function $\mathit{endpoint}_{V_1} : M_1 \mapsto V_1$. We then use the fact that edges in a matching do not share endpoints, i.e., edges in a matching are disjoint when interpreted as sets, to conclude that $\mathit{endpoint}_{V_1}$ is injective. This establishes $|M_1| \leq |V_i|$.

For $i \geq 2$ the proof of the inequality $|M_i| \leq \lfloor n_i/2 \rfloor$ is similar, but more involved. M_i is a set of edges. If we represent edges as sets, each with cardinality two, then M_i is a collection of sets. We define the set of vertices V_i' to be $\bigcup_{i\geq 2} M_i$ and use the definition of an odd-set cover to prove $V_i' \subseteq V_i$. Then, we use the fact that the edges in a matching are pairwise disjoint to prove $\left|V_i'\right| = 2 \cdot |M_i|$. Note also

type_synonym *label* = *nat*

definition *disjoint-edges* :: (α, β) *pre-graph* $\Rightarrow \beta \Rightarrow \beta \Rightarrow$ *bool* **where**
  disjoint-edges $G\ e_1\ e_2 = ($
    start $G\ e_1 \neq$ *start* $G\ e_2 \wedge$ *start* $G\ e_1 \neq$ *target* $G\ e_2 \wedge$
    target $G\ e_1 \neq$ *start* $G\ e_2 \wedge$ *target* $G\ e_1 \neq$ *target* $G\ e_2)$

definition *matching* :: (α, β) *pre-graph* $\Rightarrow \beta$ *set* $\Rightarrow$ *bool* **where**
  matching $G\ M = ($
    $M \subseteq$ *edges* $G \wedge$
    $(\forall e_1 \in M.\ \forall e_2 \in M.\ e_1 \neq e_2 \longrightarrow$ *disjoint-edges* $G\ e_1\ e_2))$

definition *OSC* :: (α, β) *pre-graph* $\Rightarrow (\alpha \Rightarrow$ *label*$) \Rightarrow$ *bool* **where**
  OSC $G\ L = ($
    $\forall e \in$ *edges* $G.$
      L (*start* $G\ e) = 1 \vee L$ (*target* $G\ e) = 1 \vee$
      L (*start* $G\ e) = L$ (*target* $G\ e) \wedge L$ (*start* $G\ e) \geq 2)$

definition *weight* :: *label set* $\Rightarrow$ (*label* $\Rightarrow$ *nat*) $\Rightarrow$ *nat* **where**
  weight $LV\ f = f\ 1 + \sum i \in LV.\ (f\ i)$ div 2

definition N :: α *set* $\Rightarrow (\alpha \Rightarrow$ *label*$) \Rightarrow$ *label* $\Rightarrow$ *nat* **where**
  $N\ V\ L\ i =$ *card* $\{v \in V.\ L\ v = i\}$

locale *matching-locale* = *digraph* +
  fixes *maxM* :: β *set*
  fixes $L :: \alpha \Rightarrow$ *label*
  assumes matching: *matching* G *maxM*
  assumes OSC: *OSC G L*
  assumes weight: *card maxM* = *weight* $\{i \in L$ ‘ *verts* $G.\ i > 1\}$ (N (*verts* G) L)

Listing 11: Definitions and locale for the matching proof in Isabelle

that $|V_i'|$ must be even since $|M_i|$ is a natural number. Thus we can prove that $|M_i| \leq \lfloor |V_i'| / 2 \rfloor$ and hence $|M_i| \leq \lfloor |V_i'| / 2 \rfloor \leq \lfloor |V_i| / 2 \rfloor = \lfloor n_i/2 \rfloor$.

Instantiating this Isabelle proof for the data structures and properties exported from VCC is mostly straightforward, since both formalizations have been chosen intentionally close to each other. We prove by induction that $N\ \{0\ ..< n\}\ L\ l$ equals \label_count(L, l, n) for every label l and that *weight* $\{2\ ..\ k\}\ f$ equals \full_weight(L, n, k) if $f\ l$ and \label_count(L, l, n) coincide. After showing that $|M|$ equals M.num_edges, we can establish the witness property for the matching checker.

5 Evaluation

We have shown that our methodology allows us to lift the trustworthiness of certifying algorithms to a new level: one can obtain formal instance correctness of certifying algorithms. A certifying algorithm returns for any input $\overline{x}$ satisfying the precondition φ a pair $(\overline{y}, \overline{w})$. It comes with an associated witness predicate $\mathcal{W}$ and an associated checker program C. If $\overline{x}$ satisfies the precondition, C is supposed to halt on input $(\overline{x}, \overline{y}, \overline{w})$ and to decide the witness predicate. We have shown in three cases that checker correctness can be established formally using VCC. Precondition plus witness predicate are supposed to imply the postcondition ψ. We have shown in the same three cases that the witness property can be established formally using Isabelle/HOL. In all cases, we followed the same approach: (1) write the checker in C and annotate it so that VCC can establish that the checker decides the witness property, (2) translate precondition, witness predicate and postcondition from VCC to Isabelle/HOL, and (3) prove the witness property in Isabelle/HOL and add the VCC-version of it as an axiom to VCC. We translated manually from VCC to Isabelle/HOL. Since the translation is purely syntactical, it could be automated.

We believe that our approach would work for all certifying algorithms discussed in the survey on certifying algorithms [23]. This might require to formalize the subject matter in Isabelle/HOL first. Our work profited from the recent formalization of graphs in Isabelle/HOL [30]. A similar effort would be needed before geometric or randomized algorithms can be tackled.

It is always a gratifying experience to see that a formalization reveals gaps in non formal reasoning. In our case, for example, we found that the checker for a maximum cardinality matching in LEDA does not verify that the computed matching M is a subgraph of G.

The matching algorithm for general graphs and its efficient implementation is an advanced topic in graph algorithms. It is a highly nontrivial algorithm which is not covered in the standard textbooks on algorithms. The following page numbers illustrate the complexity gap between the original algorithm and the checker: In the LEDA book, the description of the algorithm for computing the maximum cardinality matching and the proof of its correctness takes about 15 pages, compared to a one page description of the checker implementation.

All described theorems and lemmas have been formally verified using VCC and Isabelle/HOL. Table 1 summarizes our effort in lines of code, lines of annotations, and lines of Isabelle specifications and proofs. We observe an average ratio of 2.5 for the VCC annotation overhead which includes, besides annotations for code verification, our specifications for the witness predicates. The overhead grows to an average ratio of 6.7 when also considering our Isabelle proofs. We believe that this overhead is a small price to pay for verifying full functional correctness of code with nontrivial mathematical properties. Observe that the Isabelle linking proofs are much shorter than the high-level proofs in the Isabelle locales for the two more complex checkers. Linking proofs hence cause only a small overhead and, as we found, are mostly straighforward. In our opinion, splitting the proofs into high-level proofs for the mathematical concepts and lower-level linking proofs improved productivity and certainly helped to reduce the overall proof size and effort. The overall VCC proof time for all checkers is 10 seconds on a 2.9 GHz Intel Core i7 machine. The overall Isabelle proof time is also well below one minute on the same machine.

It took several months to develop the framework and to do the first example as described in [1]. For this paper, we reworked the framework, thereby strengthening and simplifying it at the same time. We now prove total correctness of the checkers and not only partial correctness. Moreover, we now establish that the checker accepts a triple (x, y, w) if and only if w is a valid witness for output y, whereas previously, we only proved one direction. The simplification results from dropping concrete specifications in VCC. This goes along with more succinct specifications and proofs for our original case study thanks to a new version of VCC.

	C code	VCC Annotations	Exports	Isabelle Locales	Links
Connected components	61	162	63	102	163
Shortest paths	107	318	109	279	198
Matching	124	263	78	318	164

Table 1: Lines of C code, annotations for VCC, and Isabelle declarations and proofs, excluding empty lines in all cases. The latter is split up into specifications exported from VCC, abstract proofs performed within locales, and proofs that link the concrete representation exported from VCC with the abstract proofs.

Our methodology is not confined to verifying certifying computations only. Rather, any proposition that VCC fails to establish can be exported to Isabelle and proved there. See [7, section 4.5] for a small example demonstrating this. Hence, our methodology extends the applicability of VCC and allows, in conjunction with Isabelle, verifying complicated problems that were infeasible before.

6 Related Work

The notion of a certifying algorithm is ancient. Already al-Khawarizmi in his book on algebra described how to (partially) check the correctness of a multiplication. The extended Euclidean algorithm for greatest common divisors is also certifying; it goes back to the 17th century. Yet, formal verification of checkers is recent.

In 1997, Bright et al. [11] verified a checker for a sorting algorithm that has been formalized in the Boyer-Moore theorem prover [10]. De Nivelle and Piskac formally verified the checker for priority queues implemented in LEDA [27]. Bulwahn et al. [12] describe a verified SAT checker, i.e., a checker for certificates of unsatisfiability produced by a SAT solver. They develop and prove correct the checker within Isabelle/HOL. Similar proof checkers have been formalized in the Coq [5] proof assistant [15,2]. CeTA [37], a tool for certified termination analysis, is also based on formally verified checkers, done in Isabelle/HOL. As opposed to our approach, all mentioned checkers are entirely developed and verified within the language of a theorem prover.

VCC has been applied to verify tens of thousands of C code. Popular verification targets have been restricted to system-level code from the domain of microkernels and hypervisors [4,21,35]. Our work extends the range of VCC applications to graph algorithms and in general to any code that requires nontrivial mathematical reasoning to establish full functional correctness.

In contrast to using an automatic program verifier such as VCC as we did, imperative code can also be verified in interactive theorem provers such as Coq, HOL [19], or Isabelle/HOL. This requires a formalization of the imperative language and its semantics within the theorem prover. Charguéraud describes a tool, called CFML, based on the idea of extracting characteristic formulae from programs [13]. CFML is embedded in Coq and targets imperative Caml programs. It has been applied to verify imperative data structures such as mutable lists, sparse arrays and union-find. Norrish presented a formal semantics of C formalized in the HOL theorem prover [29]. Parallel to this work, a subset of C, called C0, was formalized in Isabelle/HOL [22]. Schirmer developed a verification environment for sequential imperative programs (written in the generic imperative programming language Simpl) within Isabelle and embedded C0 into this environment [34]. His work has been applied, for instance, to verify a compiler for C0 [31]. More recently, the seL4 microkernel, written in low-level C, has been verified in Isabelle/HOL [21] based on the work of Norrish and Schirmer. The underlying approach is refinement starting from an abstract specification via an intermediate implementation in Haskell to the final C code. The reverse process can partly be automated [20] yielding a human-readable abstraction of C code in a trustable way. The latter work opens up an alternative path for the verification of certifying computations. One writes the checker in C, uses the trusted translation to obtain an equivalent Simpl program, and then proves checker correctness and the witness property in Isabelle/HOL. We plan to also try this alternative route. There are pros and cons for both approaches and it is too early to claim superiority of one approach over the other. A pro of our current approach is that we annotate the source program, a con is that we use two tools. A pro of the alternative approach is that only one tool is used, a con is that one cannot argue about the source program but must argue about a program obtained by a translation process.

Previous work that proposes, like us, to use interactive theorem provers as backends to code verification systems comprises, for instance, the link between Boogie and Isabelle/HOL [9] and the link between Why and Coq [17]. Both systems have a C verifier frontend. Usually, such approaches for connecting code verifiers and proof assistants give the latter the same information that is made available to the first-order engine, overwhelming the users of the proof assistants with a mass of detail. Instead we allow only clean chunks of mathematics to move between the verifier and the proof assistant. This hides from the proof assistant details of the underlying programming language, thus, requiring the user to discharge only interesting proof obligations.

Shortest-path algorithms, especially imperative implementations thereof, are popular as case studies for demonstrating code verification [13,28,8]. They target full functional correctness which is in contrast to our goal of instance correctness. Our approach is more general in the sense that we are not restricted to a particular implementation. Instead, our work is directly applicable to any implementation of a shortest path algorithm. Given that the implementation produces, along with the shortest paths, a witness as expected by our checker, we can guarantee the correctness of the particular output, i.e., the computed shortest paths.

To the best of our knowledge, there has been no further attempt to verify algorithms or checkers for connected components or maximum cardinality matchings beside our work.

7 Conclusion and Future Work

We described a framework for verification of certifying computations and applied it to three nontrivial combinatorial problems: connectivity of graphs, shortest paths in graphs, and maximum cardinality matchings in graphs. Our work lifts the trustworthiness of certifying algorithms to a new level.

Specifically, for each instance of the considered three problems, we can now give a formal proof of the correctness of the result. Thus, the user has neither to trust the implementation of the original algorithm nor the checker, nor does he have to understand why the witness property holds. We stress that we did not prove the correctness of the original programs, but only verify the result of their computation.

Following the lines of our work presented here, the described methodology can be applied to any other problem for which a certifying algorithm is known; see [23] for a survey. Most algorithms in LEDA [24] are certifying and, in future work, we and others plan to verify more of them, in particular, shortest paths with arbitrary edge costs (Christine Rizkallah [33]) and planarity testing (Lars Noschinski).

Our methodology is not restricted to verifying certifying computations. The integration of VCC and Isabelle/HOL should be useful whenever verification of a program requires nontrivial mathematical reasoning.

Acknowledgements We thank Ernie Cohen for his advice on VCC idioms, Mark Hillebrand for insightful discussions on VCC, and Norbert Schirmer for his initial Isabelle support. Moreover, we thank Lars Noschinski for developing a powerful graph library in Isabelle/HOL.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: Verification of certifying computations. In: Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 6806, pp. 67–82. Springer (2011)
2. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 6172, pp. 83–98. Springer (2010)
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, *Lecture Notes in Computer Science*, vol. 4111, pp. 364–387. Springer (2006)
4. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Formal verification of a microkernel used in dependable software systems. In: Computer Safety, Reliability, and Security, *Lecture Notes in Computer Science*, vol. 5775, pp. 187–200. Springer (2009)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
6. Blum, M., Kannan, S.: Designing programs that check their work. In: Symposium on Theory of Computing, pp. 86–97. ACM (1989)
7. Böhme, S.: Proving theorems of higher-order logic with SMT solvers. Ph.D. thesis, Technische Universität München (2012)
8. Böhme, S., Leino, K.R.M., Wolff, B.: HOL-Boogie—An interactive prover for the Boogie program-verifier. In: Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 5170, pp. 150–166. Springer (2008)
9. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie—an interactive prover-backend for the Verifying C Compiler. Journal of Automated Reasoning **44**(1–2), 111–144 (2010)
10. Boyer, R.S., Moore, J.S.: A theorem prover for a computational logic. In: Conference on Automated Deduction, *Lecture Notes in Computer Science*, vol. 449, pp. 1–15. Springer (1990)
11. Bright, J.D., Sullivan, G.F., Masson, G.M.: A formally verified sorting certifier. IEEE Transactions on Computers **46**(12), 1304–1312 (1997)
12. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 5170, pp. 134–149. Springer (2008)
13. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: International Conference on Functional Programming, pp. 418–430. ACM (2011)

14. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 5674, pp. 23–42. Springer (2009)
15. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Theoretical Aspects of Computing, *Lecture Notes in Computer Science*, vol. 6255, pp. 260–274. Springer (2010)
16. Edmonds, J.: Maximum matching and a polyhedron with 0,1-vertices. Journal of Research of the National Bureau of Standards **69B**, 125–130 (1965)
17. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 4590, pp. 173–177. Springer (2007)
18. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, *Lecture Notes in Computer Science*, vol. 78. Springer (1979)
19. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press (1993)
20. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 7406, pp. 99–115. Springer (2012)
21. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. Communications of the ACM **53**(6), 107–115 (2010)
22. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In: Software Engineering and Formal Methods, pp. 2–12. IEEE Computer Society (2005)
23. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011)
24. Mehlhorn, K., Näher, S.: The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
25. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
26. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *Lecture Notes in Computer Science*, vol. 2283. Springer (2002)
27. de Nivelle, H., Piskac, R.: Verification of an off-line checker for priority queues. In: Software Engineering and Formal Methods, pp. 210–219. IEEE Computer Society (2005)
28. Nordhoff, B., Lammich, P.: Dijkstra's shortest path algorithm. Archive of Formal Proofs (2012). http://afp.sourceforge.net/entries/Dijkstra_Shortest_Path.shtml, Formal proof development
29. Norrish, M.: C formalised in HOL. Ph.D. thesis, Computer Laboratory, University of Cambridge (1998)
30. Noschinski, L.: A formalization of graphs in Isabelle/HOL (2011). Ongoing work, available from author upon request
31. Petrova, E.: Verification of the C0 compiler implementation on the source code level. Ph.D. thesis, Saarland University, Saarbrücken (2007)
32. Rizkallah, C.: Maximum cardinality matching. Archive of Formal Proofs (2011). http://afp.sourceforge.net/entries/Max-Card-Matching.shtml, Formal proof development
33. Rizkallah, C.: Verification of a checker for shortest path with general edge weights (2012). Formal proof development
34. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
35. Shi, J., He, J., Zhu, H., Fang, H., Huang, Y., Zhang, X.: ORIENTAIS: Formal verified OSEK/VDX real-time operating system. In: Engineering of Complex Computer Systems, pp. 293–301. IEEE Computer Society (2012)
36. Sullivan, G.F., Masson, G.M.: Using certification trails to achieve software fault tolerance. In: Fault-Tolerant Computing, pp. 423–431. IEEE Computer Society (1990)
37. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 5674, pp. 452–468. Springer (2009)
38. Verisoft XT. http://www.verisoftxt.de (2010)