Verification of Certifying Computations

Eyad Alkassar[1], Sascha Böhme[2], Kurt Mehlhorn[3], and Christine Rizkallah[3]

[1] Universität des Saarlandes, Germany
[2] Institut für Informatik, Technische Universität München, Germany
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany

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 non-trivial 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. In this paper we develop a framework to seamlessly verify certifying computations. The automatic verifier VCC is used for checking code correctness, and the interactive theorem prover Isabelle/HOL targets high-level mathematical properties of algorithms. We demonstrate the effectiveness of our approach by presenting the verification of a typical example of the algorithmic library LEDA.

1 Introduction

One of the most prominent and costly problems in software engineering is correctness of software. In this paper, we are concerned with software for difficult algorithmic problems, e.g., matchings in 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* [3, 18, 13] produces with each output a *certificate* or *witness* that the *particular output* is correct. By inspecting the witness, the user can convince himself that the output is correct, or reject the output as buggy. Figure 1 contrasts a standard algorithm with a certifying algorithm for computing a function f.

A user of a certifying algorithm inputs x and receives the output y and the witness w. He then checks that w proves that y is a correct output for input x. The process of checking w can be automated with a *checker*, which is an algorithm for verifying that w proves that y is a correct output for x. Having checked the witness, the user may proceed with complete confidence that output y has not been compromised. Certifying algorithms are the design principle of the algorithmic library LEDA [14]: 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.

We take the principle a step further and develop a methodology for formal proofs of instance correctness. We demonstrate it on one of the more complex algorithms in

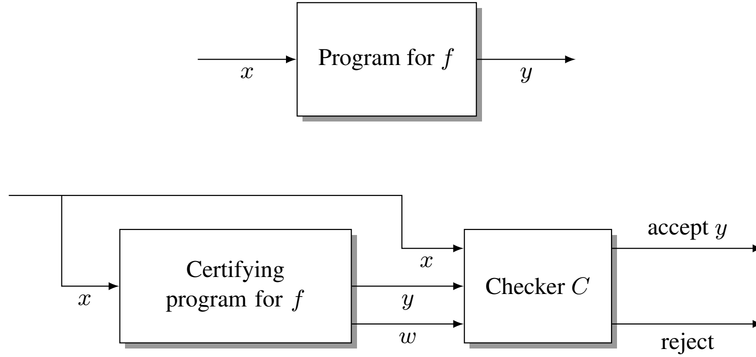

Fig. 1: The top figure shows the I/O behavior of a conventional program for computing a function f. The user feeds an input x to the program and the program returns an output y. A certifying algorithm for f computes y and a witness w. The checker C accepts the triple (x, y, w) if and only if w is a valid witness for the equality $y = f(x)$.

LEDA, maximum cardinality matching in graphs. The description of the algorithm and its implementation in [14] comprises 15 pages. In contrast, the checker is less than a page. Our formalization revealed that the checker program in LEDA is incomplete.

We outline our approach in Section 2 and give a detailed case study in Section 4. See also `http://www4.in.tum.de/~boehmes/certifying_algorithms.html` for related files. In Section 3 we survey the verfication tools VCC and Isabelle/HOL. Section 5 discusses related work and Section 6 offers conclusions.

2 Outline of Approach

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 $\psi(x, y)$. For simplicity, we only consider algorithms with trivial preconditions in this paper, i.e., $\varphi(x)$ for all $x \in X$. A *witness predicate* for a specification with postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$ with the following *witness property*

$$\mathcal{W}(x, y, w) \implies \psi(x, y) \tag{1}$$

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 $i_W : \overline{W} \to W$. We also have a concrete version $\overline{\mathcal{W}} \subseteq \overline{X} \times \overline{Y} \times \overline{W}$ of the witness predicate and a program C that checks it. The concrete version $\overline{\psi}$ of the postcondition is defined as

$$\overline{\psi}(\overline{x}, \overline{y}) \equiv \psi(i_X(\overline{x}), i_Y(\overline{y})). \tag{2}$$

We have the following proof obligations:

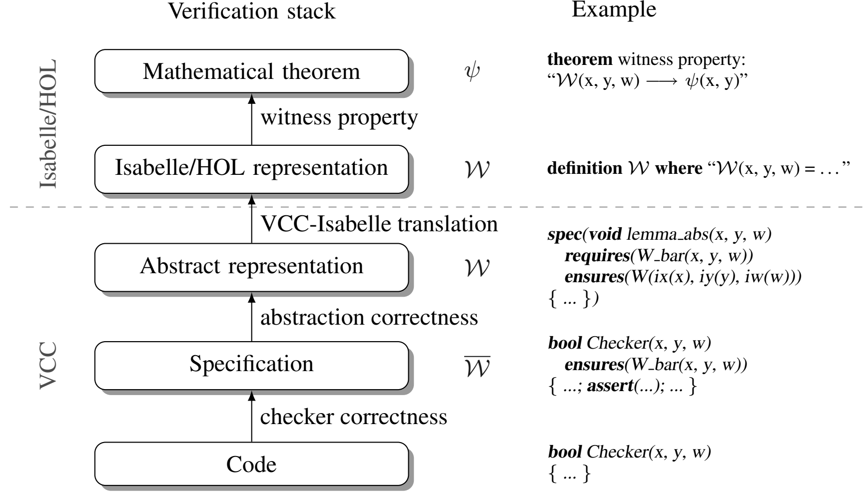

Fig. 2: Verification Framework

Checker Correctness: A formal proof that C checks $\overline{\mathcal{W}}(\overline{x}, \overline{y}, \overline{w})$, i.e., the checker C accepts $(\overline{x}, \overline{y}, \overline{w})$ if and only if $\overline{\mathcal{W}}(\overline{x}, \overline{y}, \overline{w})$ holds.

Abstraction Correctness: A formal proof of

$$\overline{\mathcal{W}}(\overline{x}, \overline{y}, \overline{w}) \implies W(i_X(\overline{x}), i_Y(\overline{y}), i_W(\overline{w})). \tag{3}$$

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

Theorem 1. *Assume that the proof obligations are fulfilled and C accepts a triple $(\overline{x}, \overline{y}, \overline{w})$. Then $\overline{\psi}(\overline{x}, \overline{y})$ by a formal proof.*

Proof. Since C accepts $(\overline{x}, \overline{y}, \overline{w})$ and we have a formal proof for the correctness of C, we have a formal proof for $\overline{\mathcal{W}}(\overline{x}, \overline{y}, \overline{w})$. By implication (3), we have a formal proof for $\mathcal{W}(i_X(\overline{x}), i_Y(\overline{y}), i_W(\overline{w}))$ and then by (1) a formal proof for $\psi(i_X(\overline{x}), i_Y(\overline{y}))$. The latter is equivalent to $\overline{\psi}(\overline{x}, \overline{y})$ by definition (2). □

We next discuss how we fulfill the various proof obligations in a *comprehensive* and *efficient* framework, see Fig. 2. Comprehensive means that the final proof formally combines (as much as possible at syntactic level) the correctness arguments for all levels (implementation, abstraction and mathematical theory). Efficient means to use the right tool for the right target. For example, applying a general theorem prover to verify imperative code would involve a lot of language-specific overhead and lead to less automatization; similarly, a specialized code verifier is often not powerful enough to cover non-trivial 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++ [14]. Our aim is to verify code which is as near 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 [6], an automatic code verifier for full C. Our choice is motivated by the maturity of the tool and the provision of a assertion language which is rich enough for our requirements. In the Verisoft XT project [20] VCC was successfully used to verify tens of thousands of non-trivial C code. VCC offers a second-order logic assertion language with ghost code and types such as maps and unbounded integers. This gives us enough expressiveness to quantify over graphs, labellings, 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 [17], due to the large set of already formalized mathematics, its descriptive proof format and its various automatic proof methods and tools. In Section 3 we overview both systems. Figure 2 shows the work-flow for verifying checkers.

Checker Verfication: Starting point is the checker code written in C. Using VCC we annotate the functions and data structures, such that the witness predicate $\overline{\mathcal{W}}$ can be established as postcondition of the checker function.

Abstraction Correctness: The witness predicate $\overline{\mathcal{W}}$ is defined over C data-structures, e.g. pointers, arrays, unions and bounded numbers. A one-to-one translation to Isabelle/HOL would have to unveil the complete type and memory axiomatization of C and VCC and would thus generate an extremely large proof context. We avoid this overhead by first abstracting all involved data structures and properties to pure mathematical objects and definitions (using VCC ghost types) by defining mappings i_X, i_Y and i_W. As a result we obtain a second-order logic formula in VCC for the witness property $\mathcal{W}$. We justify this abstraction by proving correspondence lemmas between abstract and concrete properties in VCC.

Export to Isabelle/HOL: Next—based on the abstract postcondition of the checker—we formulate the overall correctness theorem in VCC, i.e., implication (1).[4] Establishing such a theorem may involve non-trivial mathematical reasoning. Therefore we translate it to Isabelle/HOL. Due to the level of abstraction this translation is purely syntactical and does not involve any VCC specifics.

Witness Property: We prove the final theorem using Isabelle/HOL.

We stress that the overall correctness theorem is 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.

3 Tool Overview: VCC and Isabelle/HOL

VCC [6, 7, 15] is an assertional, first-order deductive code verifier for full C code. To overcome the restrictions of first-order reasoning, ghost state and code are used, e.g.,

[4] Mathematical theorems can be formulated in VCC using pure ghost functions, i.e., functions that do not alter the state.

to maintain inductively defined information. Specifications in the form of function contracts or data invariants 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) correctness, which it then tries to discharge using the Boogie verifier [2] and the automatic theorem prover Z3 [16].

Verification in VCC makes heavy use of ghost data and code (indicated by keyword ***spec()***) 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 (***mathint***), records and maps. 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 [17, 12] 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 [11], guarantees correctness as long as the inference kernel is trusted. Isabelle/HOL comes with a rich set of already formalized theories, among which are natural numbers and integers as well as sets and finite sets. New types can also be introduced. Proofs in Isabelle/HOL are written in a style close to that of mathematical textbooks. The user structures the proof and the system fills in the gaps by its automatic proof methods.

4 Case Study: Maximum Cardinality Matching in Graphs

We present a case study: maximum cardinality matchings in graphs. We obtain formal instance correctness. Our starting point is the certifying algorithm and the corresponding checker in LEDA. We give a formal proof for the correctness of the checker, for the witness property, and the connection between them.[5]

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 3 shows a graph, a maximum cardinality matching, and a witness of this fact. An *odd-set cover* OSC of a graph G is a labeling of the nodes of G with integers such that every edge of G is either incident to a node labeled 1 or connects two nodes labeled with the same number $i \geq 2$.

Theorem 2 (Edmonds [9]). *Let M be a matching in a graph G and let OSC be an odd-set cover of G. For any $i \geq 0$, let n_i be the number of nodes labeled i. If*

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor \tag{4}$$

then M is a maximum cardinality matching.

[5] All files related to our formalization can be obtained from the following URL: `http://www4.in.tum.de/~boehmes/certifying_algorithms.html`

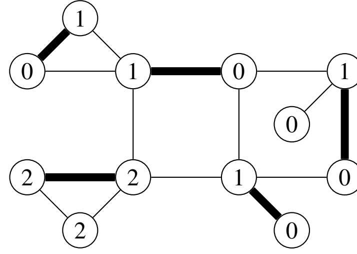

Fig. 3: The node labels certify that the indicated matching is of maximum cardinality: All edges of the graph have either both endpoints labelled as 2 or at least one endpoint labelled as 1. Therefore, any matching can use at most one edge with both endpoints labelled 2 and at most four edges that have an endpoint labelled 1. Therefore, no matching has more than five edges. The matching shown consists of five edges.

Proof. Let N be any matching in G. For $i \geq 2$, let N_i be the edges in N that connect two nodes 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

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

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

It can be shown (but this is non-trivial) that for any maximum cardinality matching M there is an odd-set cover OSC satisfying equality (4). The cover uses non-negative node 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 OSC such that (4) holds. The relationship to Section 2 is as follows:

$$\begin{aligned} X, Y &= \text{the set of all finite undirected graphs without self-loops} \\ \psi(G, M) &= M \text{ is a maximum cardinality matching in } G \\ W &= \text{odd-set covers} \\ \mathcal{W}(G, M, osc) &= M \text{ is a matching in } G,\ osc \text{ is an odd-set cover for } G, \text{ and (4) holds.} \end{aligned}$$

Theorem 2 is the witness property. We give a formal proof for it in Section 4.2. Writing a correct program which checks whether a set of edges is a matching and a node labeling is an odd-set cover which together satisfy Eq. (4) is easy. In Section 4.1 we describe the verification of such a checker. In Section 4.3, we link both results.

4.1 Checker

First, we define the specification $\overline{\mathcal{W}}$ of the checker and consider its verification against the code. Next, we abstract the postcondition to $\mathcal{W}$ and define the witness property $\mathcal{W}(x, y, w) \Longrightarrow \psi(x, y)$ which is then translated to Isabelle/HOL. Except for the witness property, which is proven in Isabelle/HOL, all presented abstractions and functions have been formally verified using VCC.

```
struct edge {unsigned s; unsigned t;};

typedef struct graph {
    unsigned m; unsigned n; // m edges and n nodes
    struct edge* es; // array of edges

    // data−type invariants
    invariant(∀(unsigned e; e < m  ⟹  es[e].s < n ∧ es[e].t < n ∧ es[e].s ≠ es[e].t))
    // further technical invariants are omitted here
} graph;
```

Listing 1: Data structures and invariants

Specification. First, we specify well-defined graphs as a property over the implementation data-type (see Listing 1). In the implementation, nodes are identified by unsigned integers, edges are represented as C structs with two components, the source and the target node, and graphs are encoded by structs with three components, numbers of nodes and edges and an edges array. In VCC we can specify data-type invariants, which are guaranteed to hold whenever an object of that data-type is ***wrapped***.[6] The graph invariant (see Listing 1) excludes self-loops and requires that endpoints of edges are in range. To establish memory safety, a set of additional invariants specifying ownership relations between different graph components are required. For convenience we have omitted them here.

Next, we specify the postcondition $\overline{\mathcal{W}}$ of the checker function. Its arguments are the original graph G, the alleged maximum matching M and two witnesses; an odd set cover *osc* and a graph embedding *id*, which are both mappings from ***unsigned*** to ***unsigned***. We specify the matching as a graph M plus an embedding *id* that maps edges of M to edges of G. Alternatively, we could have specified it as a list of edges of G. The postcondition states that the checker outputs ***true*** if and only if the following four properties hold, three of which can be expressed straightforwardly as first-oder logic formulae.

Matching: Let the ghost predicate *eadj* denote adjacent edges. Then M is a matching precisely if the following condition holds:

```
spec(ispure bool is_matching(graph* M)
  ensures(result ⟺ ∀(unsigned e1, e2; e1 < M→m ∧ e2 < M→m ∧ e1 ≠ e2 ⟹
    ¬eadj(M→es[e1], M→es[e2])));)
```

where ***ensures(...)*** defines a postcondition of a function and ***spec()*** and ***ispure()*** mark a function as ghost and free of side-effects.

Subgraph: Checking that M is a subgraph of G is made efficient by an embedding function *id*, which maps edge identifiers in M to those in G. This check is missing in the LEDA checker. Let the ghost predicate *eeq* denote equality of edges. Then subgraph is specified by:

```
spec(ispure bool is_subgraph(graph* G, graph* M, unsigned* id)
```

[6] As long as an object is wrapped, its data may not be modified. Whenever the state of an object is changed to wrapped its corresponding data-type invariants are checked.

returns(∀(**unsigned** e; e < M→m ⟹ id[e] < G→m ∧ eeq(M→es[e],
G→es[id[e]])));)

where ***returns(x)*** abbreviates ***ensures(result*** ≡ *x).*

Odd-set cover: The mapping *osc* is an odd-set cover for a graph *G* if and only if:

```
spec(ispure bool is_odd_set_cover(graph* M, unsigned* osc)
  returns( ∀(unsigned k; k < G→n ⟹ osc[k] < G→n) ∧
           ∀(unsigned e; e < G→m ⟹
             osc[G→es[e].s] ≡ 1 ∨ osc[G→es[e].t] ≡ 1 ∨
             (osc[G→es[e].t] ≡ osc[G→es[e].s] ∧ osc[G→es[e].t] > 1)));)
```

Equation (4): It states the equality of *M* →*m* and a sum. Specifying sums without using recursive functions is, however, a bit intricate.[7] Given a sum $\sum_{i<N} expr(i)$, the usual trick is to define a (finite) sequence $S[i]$ of partial sums where $S[i+1] = S[i] + expr(i)$ for $i > 0$ and 0 otherwise. The last element $S[n]$ of the sequence then defines the desired sum.

In VCC we specify the sequences of partial sums by ghost maps. Our checker has to compute (i) the sums n_i, denoting the number of nodes with label i and (ii) the overall sum. The VCC map defining n_i is specified by the equation *N[k + 1][i]* ≡ *N[k][i] + (i* ≡ *osc[k] ? 1 : 0)* with the base case *N[0][i]* ≡ *0.* The map for the overall sum is defined by the equation *SU[i + 1]* ≡ *SU[i] + N[G→n][i]/2* with the base case *SU[2]* ≡ *N[G→n][1]* and *SU[0]* ≡ *0* ∧ *SU[1]* ≡ *0* for trivial graphs. The following predicate encapsulates these conditions:

```
spec(ispure bool consistent_sums(graph* G, unsigned* osc,
     mathint N[mathint][mathint], mathint SU[mathint])
  returns(
    ∀(mathint i; 0 ≤ i ∧ i < G→n ⟹ N[0][i] ≡ 0) ∧
    ∀(mathint i, k; 0 ≤ i ∧ i < G→n ∧ 0 ≤ k ∧ k < G→n ⟹
      N[k + 1][i] ≡ N[k][i] + (i ≡ osc[k] ? 1 : 0)) ∧
    SU[0] ≡ 0 ∧ SU[1] ≡ 0 ∧ SU[2] ≡ N[G→n][1] ∧
    ∀(mathint i; 1 < i ∧ i < G→n ⟹ SU[i + 1] ≡ SU[i] + N[G→n][i]/2));)
```

Map types are declared analogously to array types, e.g., ***mathint*** *map[**mathint**]* denotes a map from unbounded integers to unbounded integers.

Based upon these definitions, Eq. (4) is formulated as *M→m* ≡ *SU[G→n].*

The complete postcondition of the checker is defined by the specification function *W_bar_holds* given in Listing 2. Note that the partial sums *N* and *SU* are passed as ghost parameters to the predicate.[8]

Implementation and Verification. The checker function is written in plain C. Its data structures have already been introduced in Listing 1. The implementation is straightforward and consists of seven loops.

[7] We do not use recursive specifications because VCC does not yet support termination proofs.

[8] Note, that the maps SU and N could be hidden by existential quantification. We have not choosen this solution due to technical problems with existential quantifiers in VCC (presumably solved soon).

```
spec(ispure bool W_bar_holds(bool checker_out, graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id,
      mathint N[mathint][mathint], mathint SU[mathint])
  ensures(
    consistent_sums(G, osc, N, SU) ∧ // sum correctly computed
    checker_out ⟺
      is_matching(M) ∧ is_subgraph(G, M, id) ∧ is_odd_set_cover(G, osc) ∧
      SU[G→n] ≡ M→m // sum equals cardinality of M
    ));)

bool max_card_match_checker(graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id
    spec(out mathint N[mathint][mathint]) spec(out mathint SU[mathint])) // ghost output
  requires(wrapped(G)) // wrapped implies that the datatype invariants hold
  requires(wrapped(M))
  requires(∀(unsigned k; k < G→n ⟹ osc[k] < G→n))
  ensures(W_bar_holds(result, G, M, osc, id, N, SU))
```

Listing 2: Implementation correctness. The VCC keywords **requires()** and **ensures()** denote pre- and postconditions of functions. The code enclosed in **spec()** is ghost code and only taken into account during verification. With **spec(out ..)** we specify ghost output variables to functions.

We verify the checker code by proving that its postcondition is equivalent to the witness predicate $\overline{\mathcal{W}}$.[9] As precondition we require that the graph and the matching are well-defined (by requiring that they are wrapped, i.e., that their data-type invariant holds), and that the odd-set cover is in range. The contract of the checker program is given in Listing 2. Note, that the partial sums *N* and *SU* are computed in ghost code and returned as ghost output values.

Most of the work in proving the postcondition lies in finding the right loop invariants. In Listing 3 we give an excerpt from the verification of the matching property. The presented code allocates memory for a local array *deg_in_M* (note, that we assume that allocation will never fail, i.e., that enough memory is available), which is used to count the number of edges adjacent to any node. We have a matching if the degree of every node is smaller than two. The proof of this fact is non-trivial and requires the use of ghost maps *w* and *b*. As we iterate over the edges of *M*, we build for each node *k* its adjacency list in *w[k]* and record for each edge *f* incident to *k* its position *b[k][f]* in the adjaceny list. The loop invariants guarantee that all edges adjacent to node *k* are stored in the sequence *w[0],...,w[deg_in_M[k]]* and that no two edges in this sequence are equal. Thus, in case *deg_in_M[k]* > 1 we have found two adjacent edges: *w[k][0]* and *w[k][1]*. In case *deg_in_M[k]* ≤ 1 we conclude that all edges adjacent to *k* are equal to *w[k][0]*, thus establishing that no two edges share the same node *k*. If *deg_in_M[k]* ≤ 1 holds for all nodes *k*, we easily can conclude that no two edges are adjacent, i.e., that *M* is a matching.

Abstraction. As preparation for the translation to Isabelle/HOL we define the checker predicate without referring to concrete C data structures. We do so by defining pure ghost data types (e.g., graphs) and corresponding abstraction functions i_X, i_Y and i_W. Moreover we prove that our abstraction is sound and complete.

[9] For soundness, implication would suffice. However, then a trivial checker returning always false would also satisfy the postcondition.

```
spec(unsigned w[unsigned][unsigned];) // w[k] ghost list of edges adjacent to k
spec(unsigned b[unsigned][unsigned];) // b[k][f] position in w of edge f adjacent to node k

unsigned* deg_in_M = malloc(M→n * sizeof (unsigned));
assume(deg_in_M ≠ NULL); // enough memory available
// zeroing
for (k = 0; k < M→n; k++)
    invariant(∀(mathint l; 0 ≤ l ∧ l < k ⟹ deg_in_M[l] ≡ 0))
    deg_in_M[k] = 0;

for(e=0; e < M→m; e++)
    invariant(M→m ≠ 0 ⟹ ∀(unsigned k, i; w[k][i] < M→m))
    invariant(∀(unsigned k, i; k < M→n ∧ i < deg_in_M[k] ⟹ w[k][i] < e))
    invariant(∀(unsigned k, i, j; k < M→n ∧ j < deg_in_M[k] ∧ i < j ⟹ w[k][i] < w[k][j]))
    invariant(∀(unsigned k; k < M→n ⟹ deg_in_M[k] ≤ e))
    invariant(∀(unsigned k, n; k < M→n ∧ i < deg_in_M[k] ⟹ adj(M→es[w[k][i]], k)))
    invariant(∀(unsigned k, f; k < M→n ∧ f < e ⟹
      (adj(M→es[f], k) ⟺ w[k][b[k][f]] ≡ f ∧ b[k][f] < deg_in_M[k])))
    // further technical invariants are omitted here
{
    spec(w[M→es[e].s][deg_in_M[M→es[e].s]] = e;)
    spec(w[M→es[e].t][deg_in_M[M→es[e].t]] = e;)

    spec(b[M→es[e].t][e] = deg_in_M[M→es[e].t];)
    spec(b[M→es[e].s][e] = deg_in_M[M→es[e].s];)

    deg_in_M[M→es[e].s]++;
    deg_in_M[M→es[e].t]++;
};

// if deg_in_M[k]>1 then we found two adjacent edges
assert(∀(unsigned k; k < M→n ∧ deg_in_M[k] > 1 ⟹ w[k][0] ≠ w[k][1] ∧ eadj(M→es[w[k][0]],
  M→es[w[k][1]])));

// if deg_in_M[k]<2 then all edges adjacent to k are equal to w[k][0]
assert(∀(unsigned k; k < M→n ∧ deg_in_M[k] < 2 ⟹ ∀(unsigned f; f < M→m ∧ adj(M→es[f],k) ⟹ f ≡
  w[k][0])));
```

Listing 3: Extract from code verification. The keyword **assert()** denotes an assertion which guides the prover. Assumptions are denoted by **assume()**.

In a naive approach one would put the coupling relation between abstraction and implementation into the data structure invariant. This, however, would make it necessary to discharge the correctness of abstraction during code verification.

Instead we chose to separate the verification of the code and the correctness of abstraction. Listing 4 presents the abstract ghost types, the abstraction functions and the lemmas establishing soundness and completeness of our abstraction. The predicate *W_holds* is derived from *W_bar_holds* by substituting arrays by maps and unsigned numbers by unbounded integers. This gives us the witness predicate $\mathcal{W}$. Using the abstract types we can finally state the overall correctness theorem (where the predicate *spec_invariants* specifies well-defined graphs):

```
spec(ispure bool final_theorem(spec_graph G, spec_graph M, funType osc, funType id)
  ensures(∀(mathint N[mathint][mathint]; ∀(mathint SU[mathint];
    W_holds(true, G, M, osc, id, N, SU) ⟹
      ∀(struct spec_graph M2; ∀(funType id2; spec_invariants(M2) ∧
          is_subgraph(G,M2,id2) ∧ is_matching(M2) ⟹ M2.m ≤ M.m)))));)
```

```
spec(
  // ghost record types instead of C structs
  struct vcc(record) spec_edge { mathint s; mathint t; };
  struct vcc(record) spec_graph { mathint m; mathint n; spec_edge es[mathint]; };

  typedef mathint funType[mathint];

  // abstraction functions (only declarations)
  ispure spec_graph abs_g(graph∗ G)
  ispure funType abs_fun(unsigned∗ id, unsigned s)

  // abstract postcondition (only declaration)
  ispure bool W_holds(bool checker_output, spec_graph G, spec_graph M, funType osc, funType id,
    mathint N[mathint][mathint], mathint SU[mathint])

  // soundness of abstraction
  ispure void lemma_sound_checker(graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id,
      mathint N[mathint][mathint], mathint SU[mathint])
    requires(wrapped(G) ∧ wrapped(M))
    requires(W_bar_holds(true, G, M, osc, id, N, SU))
    ensures(W_holds(true, abs_g(G), abs_g(M), abs_fun(osc, G→n), abs_fun(id, M→m), N, SU)) {};

  // completeness of abstraction
  ispure void lemma_complete_checker(graph∗ G, graph∗ M, unsigned∗ osc, unsigned∗ id,
      mathint N[mathint][mathint], funType SU)
    requires(wrapped(G) ∧ wrapped(M))
    requires(W_holds(true, abs_g(G), abs_g(M), abs_fun(osc, G→n), abs_fun(id, M→m), N, SU))
    ensures(W_bar_holds(true, G, M, osc, id, N, SU)) {};
)
```

Listing 4: Abstraction of postcondition. A ghost VCC record is declared with the keyword **vcc(record)**.

It states that whenever the checker returns ***true***, the given matching is maximal. Since this theorem is not referencing any C types, it can easily be translated to Isabelle/HOL.

4.2 Formal Proof of Witness Property

We explain the Isabelle proof for the witness property, i.e., Theorem 2. See Figures 4, 5, and 6 for excerpts from it. The formal proof follows the scheme of the informal proof and is split into two main parts.

For $i \geq 2$, let M_i be the edges in M that connect two nodes labeled i and let M_1 be the remaining edges in M. We use the definition of odd-set cover to prove that $M \subseteq \bigcup_{i \geq 1} M_i$ and thus $|M| \leq \sum_i |M_i|$. Let V_i be the nodes 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 odd-set cover, that every edge $e \in M_1$ has at least one endpoint in V_1. This gives rise to a function $endpoint_{V_1} : M_1 \mapsto V_1$. We then use the fact that edges in a matching do not share endpoints, i.e., are disjoint when interpreted as sets, to conclude that $endpoint_{V_1}$ 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 has cardinality equals two), then M_i is a collection of sets. We define the set of vertices V_i' to be $\bigcup M_i$ and use the definition of odd-set cover to prove $V_i' \subseteq V_i$. Then, we use the fact that the edges

```
types vertex = nat      types label = nat      types edge = (vertex × vertex)
definition finite-graph :: vertex set => edge set ⇒ bool where
 finite-graph V E ≡ finite V ∧ finite E ∧ (∀ e ∈ E. fst e ∈ V ∧ snd e ∈ V ∧ fst e ≠ snd e)
definition edge-as-set :: edge ⇒ vertex set where  edge-as-set e ≡ { fst e , snd e }
definition N :: vertex set ⇒ (vertex ⇒ label) ⇒ nat ⇒ nat where                        n_i
 N V L i ≡ card {v ∈ V. L v = i}
definition weight:: label set ⇒ (vertex ⇒ nat) ⇒ nat where  weight LV f ≡ f 1 + (∑ i∈LV. (f i) div 2)
definition OSC :: (vertex ⇒ label) ⇒ edge set ⇒ bool where
 OSC L E ≡ ∀ e ∈ E. L (fst e) = 1 ∨ L (snd e) = 1 ∨ L (fst e) = L (snd e) ∧ L (fst e) > 1
definition disjoint-edges :: edge ⇒ edge ⇒ bool where
 disjoint-edges e1 e2 ≡ fst e1 ≠ fst e2 ∧ fst e1 ≠ snd e2 ∧ snd e1 ≠ fst e2 ∧ snd e1 ≠ snd e2
definition matching :: vertex set ⇒ edge set ⇒ edge set ⇒ bool where                      M
 matching V E M  ≡ M ⊆ E ∧ finite-graph V E ∧ (∀ e1 ∈ M. ∀ e2 ∈ M. e1 ≠ e2 ⟶ disjoint-edges e1 e2)
definition matching-i :: nat ⇒ vertex set ⇒ edge set ⇒ edge set ⇒  (vertex ⇒ label) ⇒ edge set where      M_i
 matching-i i V E M L ≡ {e ∈ M. i=1 ∧ (L (fst e) = i ∨ L (snd e) = i) ∨ i>1 ∧ L (fst e) = i ∧ L (snd e) = i}
definition V-i:: nat ⇒ vertex set ⇒ edge set ⇒ edge set ⇒ (vertex ⇒ label) ⇒ vertex set where       V'_i
 V-i i V E M L ≡ ⋃ edge-as-set ‘ matching-i i V E M L
definition endpoint-inV :: vertex set ⇒ edge ⇒ vertex where
 endpoint-inV V e ≡  if fst e ∈ V then fst e else snd e
```

Fig. 4: Excerpt from the Isabelle proof: Definitions

in a matching are pairwise disjoint to prove $|V_i'| = 2 * |M_i|$. Note also 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 $|M_i| \leq \lfloor |V_i'| / 2 \rfloor \leq \lfloor |V_i| / 2 \rfloor = \lfloor n_i/2 \rfloor$.

4.3 Linking VCC and Isabelle

We have extended VCC to export purely mathematical specifications as Isabelle theories, essentially a syntactic rewriting. More precisely, VCC ghost records are translated into Isabelle records, and pure VCC ghost functions are translated into Isabelle function definitions. The former is sound and complete because the semantics of records is the same in both systems; the latter is sound and complete as we embed VCC's second-order logic into the stronger higher-order logic of Isabelle/HOL. Thereby, VCC's specification types (***bool***, ***mathint***, and map types) are mapped to equivalent Isabelle types (bool, int, and function types). Expressions of VCC comprising logical connectives, quantifiers, integer arithmetic operations, and specification functions are mapped to equivalent Isabelle terms.

Bridging the gap between the rather low-level definitions stemming from VCC and the high-level definitions from the formalization is straightforward and in large parts automatic, except for a number of cumbersome issues: (1) The VCC specification enforces a fixed numbering scheme of vertices and edges, whereas the Isabelle formalization has no such restriction – vertices are arbitrary natural numbers and edges are modelled as sets of vertices. (2) Edges of a graph and a matching in VCC do not necessarily need to be indexed by the same number, whereas in Isabelle/HOL we model a matching as a subset of a graph (which is simply a set of edges). (3) In the VCC specification, edges of a matching are not required to have the same representation as edges in the corresponding graph, i.e., sink and target vertices may be swapped. This cannot be the case in the Isabelle formalization due to the subset relationship between matchings and graphs.

```
lemma card-M1-le-NVL1:
 assumes matching V E M
 assumes OSC L E
 shows card (matching-i 1 V E M L) ≤ ( N V L 1)                         |M_1| ≤ n_1
lemma card-Mi-twice-card-Vi:
 assumes OSC L E ∧ matching V E M ∧ i > 1
 shows 2 ∗ card (matching-i i V E M L) = card (V-i i V E M L)            2 ∗ |M_i| = |V'_i|
lemma card-Mi-le-floor-div-2-Vi:
 assumes OSC L E ∧ matching V E M ∧ i > 1
 shows card (matching-i i V E M L) ≤ (card (V-i i V E M L)) div 2        |M_i| ≤ ⌊|V'_i| /2⌋
lemma card-Vi-le-NVLi:
 assumes i>1 ∧ matching V E M
 shows card (V-i i V E M L) ≤ N V L i                                    |V'_i| ≤ n_i
lemma card-Mi-le-floor-div-2-NVLi:
 assumes OSC L E ∧ matching V E M ∧ i > 1
 shows card (matching-i i V E M L) ≤ (N V L i) div 2                     |M_i| ≤ ⌊n_i/2⌋
lemma card-M-le-sum-card-Mi:
assumes matching V E M and OSC L E
shows card M ≤ (∑ i ∈ L‘V. card (matching-i i V E M L))                  |M| ≤ ∑_{i∈LV} |M_i|
theorem card-M-le-weight-NVLi:
 assumes matching V E M and OSC L E
 shows card M ≤ weight {i ∈ L ‘ V. i > 1} (N V L)                        |M| ≤ n_1 + ∑_{i≥2}⌊n_i/2⌋
theorem maximum-cardinality-matching:
 assumes matching V E M and OSC L E
 and card M = weight {i ∈ L ‘ V. i > 1} (N V L)
 shows matching V E M′ ⟶ card M′ ≤ card M                                Witness Property (Theorem 2)
```

Fig. 5: Excerpt from the Isabelle proof: Lemmas and Theorems

(4) Moreover, we require two inductive arguments for relating the VCC ghost functions *N* and *SU* with the definition of weight in Isabelle.

4.4 Evaluation

The checker in LEDA does not verify that M is a subgraph of G. This was revealed by the formalization.

The matching algorithm for general graphs and its efficient implementation is an advanced topic in graph algorithms. It is a highly non-trivial 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 ca. 15 pages, compared to a one page description of the checker implementation and a few corresponding proof lines.

All described theorems and lemmas have been formally verified, using either VCC or Isabelle/HOL. The C code of the checker, without annotations, spans 102 lines, including empty lines and sparse comments. The specification and verification adds another 318 lines for code and 245 lines for abstraction correctness. This results in a ratio of ca. 2.4 for the annotation overhead due to code verification. Overall proof time is less than 1 minute on one core of a 2.66 GHz Intel Core Duo machine.

The Isabelle theories bring in additional 632 lines of declarations and proofs for 28 lemmas and theorems. More than half of the Isabelle theories are concerned with the witness theorem (Theorem 2) and the rest links this theorem with the abstract specification exported from VCC.

```
lemma injectivity:
  assumes is-osc: OSC L E
  assumes is-m: matching V E M
  assumes e1-in-M1: e1 ∈ matching-i 1 V E M L
    and e2-in-M1: e2 ∈ matching-i 1 V E M L
  assumes diff: (e1 ≠ e2)
  shows endpoint-inV {v ∈ V. L v = 1} e1 ≠ endpoint-inV {v ∈ V. L v = 1} e2
proof −
  from e1-in-M1 have e1 ∈ M by (auto simp add: matching-i-def)
  moreover
  from e2-in-M1 have e2 ∈ M by (auto simp add: matching-i-def)
  ultimately
  have disjoint-edge-sets: edge-as-set e1 ∩ edge-as-set e2 = {}
    using diff is-m matching-disjointness by fast
  then show ?thesis by (auto simp add: edge-as-set-def endpoint-inV-def)
qed
```

Fig. 6: Excerpt from the Isabelle proof: Proof of an injectivity lemma

It took several months to develop the framework and to do the first example. Follow-up verifications will benefit from this framework.

5 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 a checker for certificates has not seen many instances so far.

Bulwahn et al [5] 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 proof assistant [8, 1]. CeTA [19], 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. This is acceptable when extending the capabilities of a theorem prover, but it is unsuitable for verifying checkers of algorithm implementations in C or similar languages.

Integrating powerful interactive theorem provers as backends to code verification systems has been exercised for VCC and Boogie with Isabelle/HOL as backend [4] as well as for Why with a Coq backend [10]. 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 only to discharge interesting proof obligations.

6 Conclusion and Future Work

We described a framework for verification of certifying computations and applied it to a non-trivial combinatorial problem: maximum cardinality matchings in graphs. Our work lifts the reliability of LEDA's maximum matching algorithm to a new level. For each instance of the maximum matching problem, 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 or the checker, nor does he have to understand why the witness property holds. We stress that we did not prove the correctness of the program, but only verify the result of the computation.

Our approach applies to any problem for which a certifying algorithm is known; see [13] for a survey. Most algorithms in LEDA [14] are certifying and, in future work, we plan to verify all of them. The checkers and the proof of the witness properties for all other graph algorithms in LEDA are simpler than the presented one and hence will proceed analogously.

Our framework is not only applicable to verifying certifying computations. The integration of VCC and Isabelle/HOL should be useful whenever verification of a program requires non-trivial mathematical reasoning.

Acknowledgment. We thank Ernie Cohen for his advice on VCC idioms and Norbert Schirmer for his initial Isabelle support.

References

1. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Interactive Theorem Proving, LNCS, vol. 6172, pp. 83–98. Springer (2010)
2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO 2005. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2006)
3. Blum, M., Kannan, S.: Designing programs that check their work. In: Proceedings of the 21th Annual ACM Symposium on Theory of Computing (STOC'89). pp. 86–97 (1989)
4. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie — An interactive prover-backend for the Verifying C Compiler. J. Automated Reasoning 44(1–2), 111–144 (2010)
5. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, LNCS, vol. 5170, pp. 134–149. Springer (2008)
6. 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 (TPHOLs 2009). LNCS, vol. 5674, pp. 23–42. Springer (2009)
7. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Computer Aided Verification (CAV 2010). LNCS, vol. 6174, pp. 480–494. Springer (Jul 2010)
8. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: Theoretical Aspects of Computing – ICTAC 2010, LNCS, vol. 6255, pp. 260–274. Springer (2010)
9. Edmonds, J.: Maximum matching and a polyhedron with 0,1 - vertices. Journal of Research of the National Bureau of Standards 69B, 125–130 (1965)

10. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) Conference on Computer Aided Verification. LNCS, vol. 4590, pp. 173–177. Springer (2007)
11. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, Lecture Notes in Computer Science, vol. 78. Springer (1979)
12. Gordon, M.J.C., Pitts, A.M.: The HOL logic and system. In: Towards Verified Systems, Real-Time Safety Critical Systems Series, vol. 2, chap. 3, pp. 49–70. Elsevier (1994)
13. McConnell, R., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review In Press, Corrected Proof (2010)
14. Mehlhorn, K., Näher, S.: The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
15. Microsoft Corp.: VCC: A C Verifier. http://vcc.codeplex.com/ (2009)
16. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
17. 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)
18. Sullivan, G., Masson, G.: Using certification trails to achieve software fault tolerance. In: Randell, B. (ed.) Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS '90). pp. 423–433. IEEE (1990)
19. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Theorem Proving in Higher Order Logics, LNCS, vol. 5674, pp. 452–468. Springer (2009)
20. Verisoft XT, http://www.verisoft.de/index_en.html