An Introduction to Certifying Algorithms

Zertifizierende Algorithmen: Eine Einführung

Eyad Alkassar, Saarland University, Saarbrücken,
Sascha Böhme, Technische Universiät München, Garching,
Kurt Mehlhorn, Christine Rizkallah, Max-Planck-Institut für Informatik, Saarbrücken,
Pascal Schweitzer, The Australian National University, Canberra, Australia

Summary A certifying algorithm is an algorithm that produces, with each output, a certificate or witness that the particular output is correct. A user of a certifying algorithm inputs *x*, receives the output *y* and the certificate *w*, and then checks, either manually or by use of a program, that *w* proves that *y* is a correct output for input *x*. In this way, he/she can be sure of the correctness of the output without having to trust the algorithm. We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms, and that for complex algorithmic tasks, only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The widespread use of certifying algorithms would greatly enhance the reliability of algorithmic software. We also demonstrate that the formal verification of result checkers is within the reach of current verification technology. The combination of certifying algorithms and formal verification of result checkers leads to formally verified computations. ►►► **Zusammenfassung** Ein zertifizierender Algorithmus liefert zusätzlich zur gewöhnlichen Ausgabe ein Zertifikat, welches beweist, dass die gelieferte Ausgabe korrekt ist. Ein Benutzer des zertifizierenden Algorithmus erhält bei Eingabe *x* nicht nur eine Ausgabe *y*, sondern zusätzlich das Zertifikat *w*. Der Benutzer überprüft nun, entweder von Hand oder mit Hilfe eines Programmes, ob das Zertifikat *w* tatsächlich beweist, dass *y* eine korrekte Ausgabe bei Eingabe *x* ist. Der Benutzer kann dadurch der Korrektheit der Ausgabe sicher sein, selbst wenn er der Korrektheit des Algorithmus misstraut. Wir behaupten, dass zertifizierende Algorithmen nicht-zertifizierenden Algorithmen weit überlegen sind, und dass für komplexe algorithmische Berechnungen nur zertifizierende Algorithmen adäquat sind. Daher sollte der Begriff Teil der Informatikausbildung sein und als Prinzip des Algorithmenentwurfs gelehrt werden. Die konsequente Verwendung von zertifizierenden Algorithmen erhöht die Verlässligkeit von Software deutlich. Wir zeigen, dass mit heutiger Verifikationstechnologie das Verifizieren von Ergebnischeckern im Bereich des Möglichen liegt. Die Kombination von zertifizierenden Algorithmen und formaler Verfikation ermöglicht formal verifizierte Berechnungen.

Keywords D.2.5 [Software: Software Engineering: Software/Program Verification]; D.2.5 [Software: Software Engineering: Testing and Debugging]; certifying algorithms ►►► **Schlagwörter** Software Engineering, Programmverifikation, zertifizierende Algorithmen

1 Introduction

One of the most prominent and costly problems in software engineering is correctness of software. When the user gives *x* as an input and the program outputs *y*, the user usually has no way of knowing whether *y* is a correct output on input *x* or whether it has been compromised by a bug. The user is at the mercy of the program. A *certifying algorithm* is an algorithm that produces with each

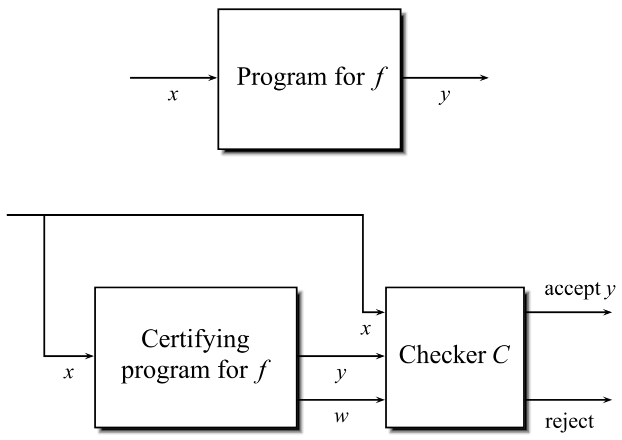


Figure 1 The top figure shows the I/O behavior of a conventional program for computing a function f. The user feeds an input x to the program and the program returns an output y. The user of the program has no way of knowing whether y is equal to $f(x)$. The bottom figure shows the I/O behavior of a certifying algorithm, which 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 $y = f(x)$.

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. He is no longer on the mercy of the program. 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. In many cases, the checker is so simple that a trusted implementation of it can be produced. Formal verification of complex algorithms is not feasible with current verification technology. However, formal verification of checkers is feasible with current technology. Assume that we have formally proved that if the checker C accepts the triple (x, y, w) then $y = f(x)$. Assume also that C accepts a triple (x, y, w). Then we have a formal proof for the correctness of y and the user may proceed with complete confidence that output y has not been compromised.

A tutorial example is the problem of recognizing whether a graph is bipartite, that is, whether the vertices can be colored with colors red and blue such that all edges have distinctly colored endpoints. A non-certifying algorithm for testing bipartiteness outputs a single bit y; the bit tells whether the graph is bipartite or not. A certifying algorithm does more; it proves its answer correct by providing an appropriate witness w, see Fig. 2. If the graph is bipartite, then w is a proper coloring. If the graph is not bipartite, w is an odd-length cycle in G. Clearly odd-length cycles are non-bipartite. Since a two-coloring of G would induce a two-coloring of the odd cycle contained in G, G cannot be two-colorable.

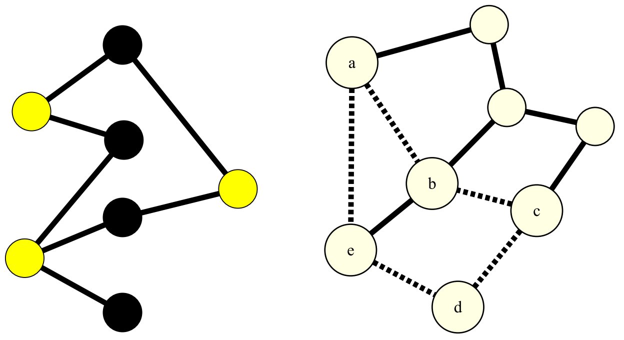


Figure 2 The graph on the left is bipartite and the coloring of the nodes proves this fact. The graph on the right is non-bipartite. The highligthed edges form an odd-length cycle and odd-length cycles are non-bipartite. Assume for example vertex a is colored red. Then b must be colored blue, then c must be colored red, then d must be colored blue, then e must be colored red, and hence a must be colored blue, a contradiction.

We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms and that for complex algorithmic tasks only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The wide-spread use of certifying algorithms would greatly enhance the reliability of algorithmic software. Certifying algorithms are available for many algorithmic tasks, see [14] for a survey, and are the design principle for LEDA, the library of efficient data types and algorithms [13; 15]. Formal verification of checkers is within reach of current verification technology, and thus allows to formally verify the correctness of outputs of complex certifying algorithms.

Certifying algorithms have many advantages over standard algorithms. They can be tested on every input, they are reliable in the sense that errors are immediately caught, they allow to trust a program without understanding it or even without knowing it, and they greatly increase the reliability of implementations. The quality of the programs in LEDA rests to a large extent on the consequent use of certifying algorithms.

We follow a simple plan. We discuss two case studies and then survey history, extant work, and the relation to testing and verification.

2 Case Study I: Planarity of Graphs

Graphs are frequently used to model and visualize relationships between entities, e. g., vertices represent companies and edges represent partial ownership. Illustrative visualizations of graphs can be obtained by drawing the graphs with no or few edge crossings. This is one of the many reasons, why planar graphs are highly relevant in various contexts.

A *planar embedding* of an undirected graph G is a drawing of the graph in the plane such that no two edges of G cross. The graphs in Figs. 2 and 4 show examples of this. A graph is *planar* if it is possible to embed it in the plane in this way. Planar graphs were among

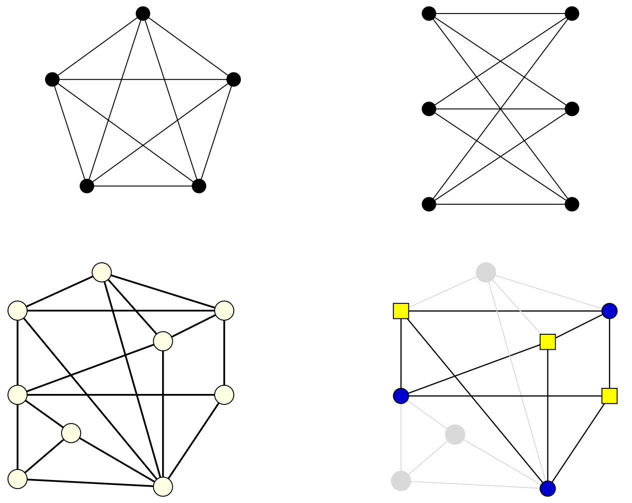


Figure 3 Every non-planar graph contains a subdivision of one of the depicted Kuratowski graphs K_5 and $K_{3,3}$ as a subgraph. We invite the reader to convince herself/himself that K_5 and $K_{3,3}$ are non-planar. Take a piece of paper and try to draw these graphs in a planar way. You will get stuck. Convince yourself that you will always get stuck, no matter how hard you try. We suggest that you first draw a cycle passing through all the vertices and then try to add the other edges. The lower part of the figure shows a non-planar graph. Non-planarity is witnessed by a $K_{3,3}$.

the first classes of graphs that were studied extensively. In the early 1900's, there was an extensive effort to give a *characterization* of those graphs that are planar. In 1930, Kuratowski gave what remains one of the most famous theorems in graph theory [12]: a graph is planar if and only if it has no *subdivision* of a K_5 or a $K_{3,3}$ as a subgraph (see Fig. 3). The K_5 is the complete graph on five vertices, the $K_{3,3}$ is the complete bipartite graph with three vertices in each bipartition class, and a subdivision of a graph is what is obtained by repeatedly subdividing edges by inserting vertices of degree two on them.

The planarity test in LEDA played a crucial role in the development of certifying algorithms. There are several efficient algorithms for planarity testing. An implementation of the Hopcroft and Tarjan [11] algorithm was added to LEDA in 1991. The implementation had been tested on a small number of graphs. In 1993, a researcher sent Mehlhorn and Näher a graph together with a planar drawing of it, and pointed out that the program declared the graph non-planar. It took Mehlhorn and Näher some days to discover the bug. More importantly, they realized that a complex question of the form "is this graph planar" deserves more than a yes-no answer. They adopted the thesis that

a program should justify (prove) its answers in a way that is easily checked by the user of the program.

What does this mean for the planarity test? If a graph is declared planar, a proof should be given in the form of a planar drawing or an embedding, which the program already did. If a graph is non-planar, the program should not just declare this; it should supply a proof of this.

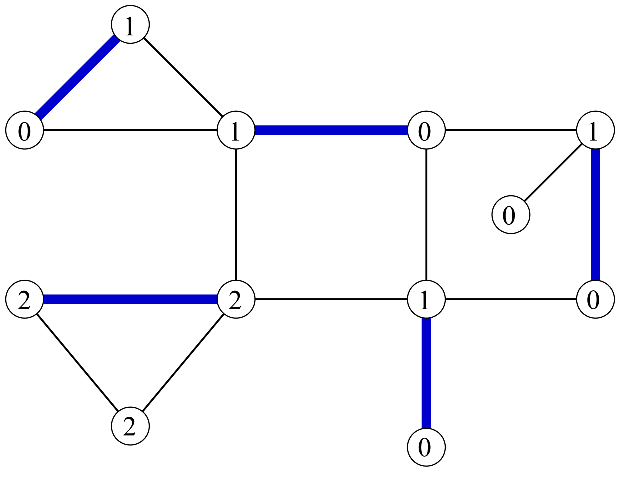


Figure 4 The node labels certify that the indicated matching is of maximum cardinality: all edges in the graph have either both endpoints labeled as two or at least one endpoint labeled as one. Since a matching is a subgraph, this statement also holds for the edges in any matching. Therefore, any matching can use at most one edge with both endpoints labeled two and at most four edges that have an endpoint labeled one. Therefore, no matching has more than five edges. The matching shown consists of five edges and hence has maximum cardinality.

The existence of an obvious proof is known by Kuratowski's theorem: it suffices to point out a Kuratowski subgraph. Subsequently an efficient algorithm for finding Kuratowski subgraphs was added to the library, which makes the planarity test a certifying algorithm.

The user of the certifying planarity test only needs to understand the easy direction of Kuratowski's theorem, namely, that graphs that contain a subdivision of a K_5 or $K_{3,3}$ are non-planar. There is no need to understand the difficult direction that non-planar graphs always contain a subdivision of a K_5 or a $K_{3,3}$.

3 Case Study II: Maximum Cardinality Matchings in Graphs

Assume you run a dating service. You have clients and for each client you have a list of other clients with which you might match with the client. This situation is readily modeled as a graph. There is a vertex for each client and an edge for each possible match.[1] The goal of the dating service is to arrange a maximum number of dates. This is a maximum cardinality matching problem.

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 4 shows a graph, a matching, and a proof that the matching has maximum cardinality. Such a proof always exists. An *odd-set cover* *OSC* of G is a labeling of the vertices of G with nonnegative integers such that every edge of G is either incident to a vertex labeled 1 or connects two vertices labeled with the same number $i \geq 2$.

[1] A dating service will also have an estimate of success for each possible match; this makes the problem harder. Also, it will typically match clients of different sex; this makes the problem simpler.

Theorem 1 ([7]). *Let M be any matching in G and let OSC be an odd-set cover of G. For any $i \geq 0$, let n_i be the number of vertices labeled i. If*

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor \,, \qquad (1)$$

M is a maximum cardinality matching.

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

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

Thus $|N| = |N_1| + \sum_{i \geq 2} |N_i| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor = |M|$.

We refer to the hypothesis of Theorem 1 "M is a matching in G, OSC is an odd-set cover for G, and Eq. (1) holds" as *witness predicate*. It can be shown (but this is non-trivial) that for any maximum cardinality matching M there is an odd-set cover OSC such that Eq. (1) holds and all labels are integers between 0 and $n-1$, where n is the number of vertices of G. In such a cover all n_i with $i \geq 2$ are odd, hence the name. A *certifying algorithm for maximum cardinality matching* returns a matching M and an odd-set cover OSC such that (1) holds. Edmonds designed a certifying algorithm for maximum cardinality matchings. An efficient implementation of the algorithm is part of LEDA.

Let us next discuss a checker. Its input is the graph $G = (V, E)$, a list M of edges of G and an odd-set cover OSC. In order to verify the witness predicate, the checker performs the following tests:

1. Check that M is a matching, i. e., that no two edges in M share an endpoint. In order to perform this check efficiently, we determine for each vertex of G, its degree with respect to M. We initialize an array indexed by vertices to zero and then iterate over the edges in M. For an edge $(s,t) \in M$ we increase the degree of s and t. If the degrees of all vertices stay below two, M is a matching.
2. Check that OSC is a vertex cover. We iterate over all edges (s,t) of G and check that either s or t is labeled 1 or that both have the same label and this label is greater than one. We also check that all labels are between 0 and $n-1$ inclusive.
3. Finally, we check Eq. (1).

A program for the above is short and simple, see Fig. 5.

However, even short and simple programs may be incorrect. In fact in the version of the checker in [15],

```
static bool False(string s)
{ cerr << "CHECK_MAX_CARD_MATCHING: " << s << "\n"; return false; }

bool CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                                    const node_array<int>& OSC)
{ int n = Max(2,G.number_of_nodes());
  array<int> count(n);
  for (int i = 0; i < n; i++) count[i] = 0;
  node v; edge e;

  node_array<int> deg(G,0);
  forall(e,M) { deg[G.source(e)]++; deg[G.target(e)]++; }
  forall_nodes(v,G) if (deg[v] > 1) return False("M is not a matching");

  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if ( v == w || OSC[v] == 1 || OSC[w] == 1 ||
            ( OSC[v] == OSC[w] && OSC[v] >= 2) ) continue;
    return False("OSC is not a cover");
  }

  forall_nodes(v,G)
  { if ( OSC[v] < 0 || OSC[v] >= n )
      return False("negative label or label larger than n - 1");
    count[OSC[v]]++;
  }

  int S = count[1];
  for (int i = 2; i < n; i++) S += count[i]/2;
  if ( S != M.length() ) return False("OSC does not prove optimality");

  return true;
}
```

Figure 5 The checker for maximum cardinality matchings.

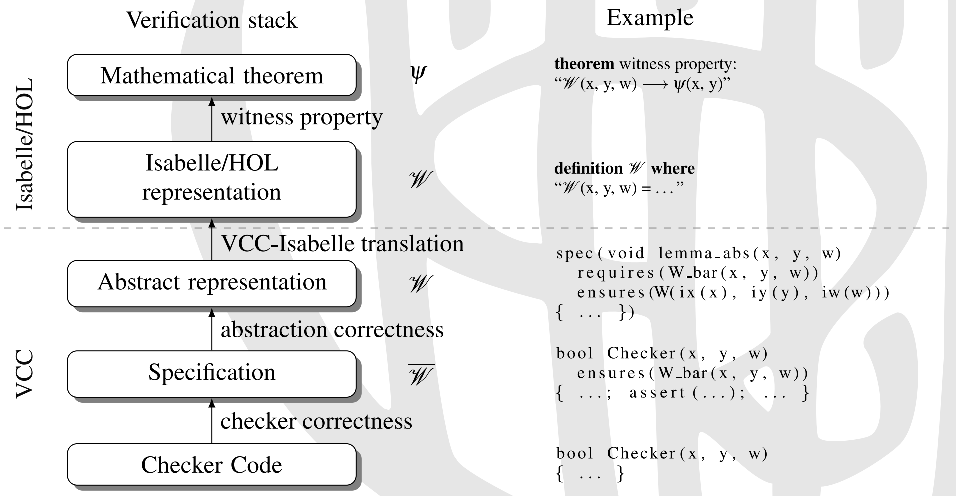
the check whether M is a matching is missing. This is where formal verification steps in. Assume we had formal proofs of the following three statements:

- **Checker Correctness**: The program of Fig. 5 accepts a triple (G, M, OSC) if and only if M is a matching in G, OSC is an odd-set cover of G, and Eq. (1) holds.
- **Witness Property**: Theorem 1.
- **Abstraction Correctness**: Theorem 1 refers to mathematical concepts, such as graphs and functions from vertices to integers. The input to the checker are a concrete representation of a graph, a list M of edges, and an array OSC indexed by vertices. So we need a proof that what the checker checks for the concrete objects is what Theorem 1 requires for the abstract objects.

Then we have formal proofs of instance correctness, i. e., whenever a triple (G, M, OSC) is accepted by our checker, we have a formal proof that M is a maximum cardinality matching in G. We still have no guarantee that our matching algorithm is correct. However, we have the next best situation. If a triple (G, M, OSC) is accepted by our checker, we have a formal proof that M is of maximum cardinality, and if the triple is not accepted, we are pointed to a bug. If the program is correct, the latter will never happen. If the program is not correct, errors will not go unnoticed.

In [2] we have carried out the approach outlined above after reformulating the checker in the programming language C. Figure 6 shows the workflow. We verify code with VCC [6], an automatic code verifier for the entire language C. Our choice is motivated by the maturity of the tool and the availability of an assertion language which is rich enough for our requirements. In the Verisoft XT project [18] VCC was successfully used to verify tens of thousands lines 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]. We do so, since it has a large set of already formalized mathematics, a descriptive proof format and various automatic proof methods and tools. We next discuss how we resolved the three proof obligations mentioned above.

- **Checker Correctness**: Starting point is the checker code written in C. Using VCC we annotate the functions and data structures, such that the concrete witness predicate can be established as postcondition of the checker function.
- **Abstraction Correctness**: The concrete witness predicate 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 suitable abstraction mappings. As a result we obtain a second-order logic formula in VCC for the abstract witness property. 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 Theorem 1 in VCC. 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.
- **Witness Property**: We prove the final theorem using Isabelle/HOL.



theorem witness property:
"$\mathscr{W}$(x, y, w) $\longrightarrow$ ψ(x, y)"

definition $\mathscr{W}$ **where**
"$\mathscr{W}$(x, y, w) = . . ."

```
spec(void lemma_abs(x, y, w)
  requires(W_bar(x, y, w))
  ensures(W(ix(x), iy(y), iw(w)))
{ ... })

bool Checker(x, y, w)
  ensures(W_bar(x, y, w))
{ ...; assert(...); ... }

bool Checker(x, y, w)
{ ... }
```

Figure 6 The workflow of the verification: The checker is written in C. We prove in VCC that it meets its specification, i. e., accepts a triple of (representation of a graph, list of edges of the graph, labeling of the nodes of the graph) if and only if the triple satisifies the concrete version of the witness predicate. We also formulate the abstract version of the witness predicate in VCC and prove its equivalence to the concrete version. The abstract version of the witness predicate is translated to Isabelle/HOL and Theorem 1 is proved there.

4 History and Extant Work

The notion of a certifying algorithm is ancient. Already al-Khwarizmi 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. The seminal work by Blum and Kannan [3] on programs that check their work put result checking in the limelight. There is however an important difference between certifying algorithms and programs that check their work. Certifying programs produce with each output an easy-to-check certificate of correctness; Blum and Kannan are mainly concerned with checking the work of programs in their standard form. Mehlhorn and Näher adopted certifying algorithms as a design principle for LEDA.

How general is the approach? The pragmatic answer is that we know of 100+ certifying algorithms, see [14] for a survey. In particular, there are certifying algorithms for the problems that are usually treated in an introductory algorithms course, such as connectedness and strong connectedness, minimum spanning trees, shortest paths, maximum flows, and maximum matchings. The theoretical answer is that every algorithm can be made weakly certifying[2] without asymptotic loss of efficiency, but the construction underlying this result is artificial and requires a correctness proof in some formal system. However, the result is also assuring: certifying algorithms are not elusive. The challenge is to find natural ones.

The two main approaches to program correctness are program testing and program verification. *Program testing* [19] executes a given program on inputs for which the correct output is already known by some other means, e. g., by human effort or by execution of another program that is believed to be correct. However, in most cases, it is infeasible to determine the correct output for all possible inputs by other means or to test the software on all possible inputs. Thus testing is usually incomplete and therefore bugs may evade detection. The Pentium bug is an example [4]. Certifying programs greatly enhance the power of testing. A certifying program can be tested on every input. The test is whether the checker accepts the triple (x, y, w). If it does not, either output or witness is incorrect.

Program verification refers to formal proofs of correctness of programs. The principles are well established [8; 10]. However, handwritten proofs are only possible for small programs owing to the complexity and tediousness of the proof process. Using computer-assisted proof systems, formal proofs for interesting theorems were recently given [9; 16; 18]. Program verification and certifying algorithms can support each other. Checkers are usually much simpler than the algorithms they check; they are amenable to formal verification as we have seen in Sect. 3; the paper [5] provides an earlier example.

[2] A weakly certifying algorithm is only required to halt for inputs satisfying the precondition. The witness proves that either the input did not satisfy the precondition or the output satisfies the postcondition, but it does not tell which alternative holds.

5 Conclusions

We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms and that for complex algorithmic tasks only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The wide-spread use of certifying algorithms would greatly enhance the reliability of algorithmic software. Certifying algorithms are available for many algorithmic tasks and are the design principle for LEDA. Formal verification of checkers is within reach of current verification technology. Certifying algorithms have many advantages over standard algorithms. They can be tested on every input, they are reliable in the sense that errors are immediately caught, they allow to trust a program without understanding it or even without knowing it, and they greatly increase the reliability of implementations.

References

[1] Link to Robert of Chester's Latin Translation of the Algebra of al-Khowarizmi: http://library.albany.edu/preservation/brittle_bks/khuwarizmi_robertofchester/Abbrev.pdf.

[2] E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. Verification of certifying computations. To appear in CAV (Computer Aided Verfication) 2011. Available at the third author's webpage http://www.mpi-inf.mpg.de/ mehlhorn/ftp/VerificationCertComps.pdf.

[3] M. Blum and S. Kannan. Designing programs that check their work. In: Journal of the ACM, 42(1):269–291, 1995. Preliminary version in STOC'89.

[4] M. Blum and H. Wasserman. Reflections on the Pentium division bug. In: IEEE Trans. on Computing, 45(4):385–393, 1996.

[5] J. D. Bright, G. F. Sullivan, and G. M. Masson. A formally verified sorting certifier. In: IEEE Trans. on Computers, 46(12):1304–1312, 1997.

[6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics (TPHOLs 2009), LNCS 5674, pages 23–42. Springer, 2009.

[7] J. Edmonds. Maximum matching and a polyhedron with 0,1 – vertices. In: Journal of Research of the National Bureau of Standards, 69B:125–130, 1965.

[8] R. Floyd. Assigning meaning to programs. In: J. T. Schwarz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967.

[9] G. Gonthier. Formal proof – the Four-Color theorem. In: Notices of the American Mathematical Society, 55(11):1382–1393, 2008.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. In: Communications of the ACM, 12(10):576–585, 1969.

[11] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. In: Journal of the ACM, 21(4):549–568, 1974.

[12] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, vol. 15, pages 271–283, 1930.

[13] LEDA (Library of Efficient Data Types and Algorithms). www.algorithmic-solutions.com.

[14] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. In: Computer Science Review, 5(2):119–161, 2011.

[15] K. Mehlhorn and S. Näher. The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press, 1999.
[16] J. Strother Moore and Q. Zhang. Proof pearl: Dijkstra's shortest path algorithm verified with ACL2. In: Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS 3603, pages 373–384. Springer, 2005.
[17] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS 2283. Springer, 2002.
[18] Verisoft XT. http://www.verisoft.de/index_en.html.
[19] A. Zeller. WHY PROGRAMS FAIL: A Guide to Systematic Debugging. Morgan-Kaufmann, 2005.

Received: March 4, 2011

Dr.-Ing. Eyad Alkassar received his PhD from Saarland University on formal systems verification. From 2002–2010 his research was focused on the Verisoft projects, aiming at the formal verification of entire computer systems. Currently, holding a postdoctoral position at the MMCI Cluster of Excellence, Saarland University, his main research interests are memory virtualization, kernel correctness and methods for concurrent code verification.

Address: MMCI Cluster of Excellence – Saarland University, Campus E 1 3, 66123 Saarbrücken, Germany, Tel.: +49-681-302-5557, Fax: +49-681-302-4290, e-mail: eyad@wjpserver.cs.uni-saarland.de

Dipl.-Inf. Sascha Böhme is a PhD student under the supervision of Prof. Tobias Nipkow at Technische Universität München since 2007. Prior to this, he graduated in computer science at Technische Universität Dresden.

Address: Institut für Informatik, Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany, Tel.: +49-89-28917330, Fax: +49-89-28917307, e-mail: boehmes@in.tum.de

Prof. Dr. Kurt Mehlhorn is director at the Max Planck Institute for Informatics. He is author of six books and more than 200 research articles. He is recipient of the Leibniz, Beckurts, and EATCS awards and an ACM fellow.

Address: Max-Planck-Institut für Informatik, Campus E1 4, 66132 Saarbrücken, Germany, Tel.: +49-681-9325-100, Fax: +49-681-9325-199, e-mail: mehlhorn@mpi-inf.mpg.de

M.Sc. Christine Rizkallah is a PhD student at the Max Planck Institute for Informatics. Prior to this, she received a master's degree from Saarland University.

Address: Max-Planck-Institut für Informatik, Campus E1 4, 66132 Saarbrücken, Germany, Tel.: +49-681-9325-114, Fax: +49-681-9325-199, e-mail: crizkall@mpi-inf.mpg.de

Dr. Pascal Schweitzer received a PhD from the Max Planck Institute for Informatics at Saarland University. He is a postdoctoral fellow at the Australian National University supported by the national research fund of Luxembourg.

Address: The Australian National University, North Road Building 108, 0200 Canberra, Australia, Tel.: +61-2-61259665, Fax: +61-2-61250010, e-mail: pascal.schweitzer@anu.edu.au