Verification of Certifying Computations through AutoCorres and Simpl

Lars Noschinski[1], Christine Rizkallah[2], and Kurt Mehlhorn[2] *

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

Abstract. Certifying algorithms compute not only an output, but also a witness that certifies the correctness of the output for a particular input. A checker program uses this certificate to ascertain the correctness of the output. Recent work used the verification tools VCC and Isabelle to verify checker implementations and their mathematical background theory. The checkers verified stem from the widely-used algorithms library LEDA and are written in C. The drawback of this approach is the use of two different tools. The advantage is that it could be carried out with reasonable effort in 2011. In this article, we evaluate the feasibility of performing the entire verification within Isabelle. For this purpose, we consider checkers written in the imperative languages C and Simpl. We re-verify the checker for connectedness of graphs and present a verification of the LEDA checker for non-planarity of graphs. For the checkers written in C, we translate from C to Isabelle using the AutoCorres tool set and then reason in Isabelle. For the checkers written in Simpl, Isabelle is the only tool needed. We compare the new approach with the previous approach and discuss advantages and disadvantages. We conclude that the new approach provides higher trust guarantees and it is particularly promising for checkers that require domain-specific reasoning.

1 Introduction

A user of a program has in general no easy means to know whether the result computed by the program is correct or has been compromised by a bug. While formal verification is one solution, for complex programs the cost is often prohibitive. We are interested in complex programs for combinatorial and geometric problems as, for example, discussed in [1, 4, 21]. For an input x, a *certifying algorithm* [7, 26, 19] produces an output y and a *witness* w. The accompanying *checker* is a simpler and more efficient program that uses the witness w to ascertain that y is a correct output for input x. The checker is supposed to return $true$ if and only if the witness w indeed proves that y is the correct output for x. A small example helps understanding the concept. The input for a planarity test is a graph. A certifying planarity test witnesses the output "is-planar" by a planar embedding and the output "is-not-planar" by a Kuratowski subgraph. Certifying algorithms are a key design principle of the algorithms library LEDA [21]. Checkers are an integral part of the library and are optionally invoked after every execution of a LEDA program. Adoption of the principle greatly improved the reliability of the library [20].

* The first two authors contributed equally to this work. The third author supervised the work.

The (relative) simplicity of checkers makes them amenable to formal verification. Recent work [2] provides a framework for verifying certifying computations. The approach uses the interactive theorem prover Isabelle as a backend to the automatic code verifier VCC. Low level properties of the C code are proven using VCC. These are then translated to Isabelle and used to derive the desired mathematical properties, which are translated back to VCC. This framework (the *VCC approach*) is illustrated on several examples in the domain of graphs. Using two proof tools has the advantage of using the strength of each tool: verification of C code with VCC and mathematical reasoning with Isabelle/HOL.

In this work, we investigate the feasibility of carrying out the entire verification of the checkers within Isabelle/HOL. We implement the checkers both in Simpl and in C. Simpl [25] is a generic imperative programming language embedded into Isabelle/HOL that was designed as an intermediate language for program verification. The Simpl checkers are verified directly within Isabelle. To translate from C to Isabelle we use the C-to-Isabelle parser that was developed as part of the seL4 project [17] and was used to verify a full operating system kernel. We do not work on the output of the parser directly, but use the AutoCorres tool [15] that simplifies reasoning about C in Isabelle/HOL. This approach (the *AutoCorres approach*) avoids double formalizations in two systems and reduces the trusted code base: instead of trusting VCC, one now has to trust the C-to-Isabelle parser, a significantly simpler program. Since we are the first external users of AutoCorres, it was not clear at the beginning of our work, whether the AutoCorres approach is competitive. At least for our examples, it is competitive, if not superior.

Why do we verify implementations both in C and in Simpl? It allows us to separate the verification of the checker algorithm and of the checker implementation. Simpl has a very powerful expression language as all Isabelle expressions are Simpl expressions. Therefore, one can write pseudo-code like Simpl programs. Verifying both a C and a Simpl implementation allows us to estimate how much additional effort for the full verification is needed in addition to the pseudo-code verification. The hope was that after the Simpl verification is done the verification of the C-program would be only dealing with C-intricacies and hence be relatively straight-forward.

Section 3 introduces the implementations and verifications of the checkers both in Simpl and in C and discusses lessons learned. In Section 4 we suggest a refinement framework for using Autocorres. Then in Sections 5 and 6 we give an evaluation and talk about related work. The full implementation and all proofs are available on the companion website.[3]

2 Preliminaries

As in the VCC approach, we consider algorithms taking an input x from a set X and producing an output y from a set Y and a witness w from a set W. Input x is supposed to satisfy a precondition $\varphi(x)$, and x and y are supposed to satisfy a postcondition $\psi(x, y)$. A *witness predicate* for a specification with precondition φ and postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$ with the following *witness property*:

[3] `http://www21.in.tum.de/~noschinl/Verifying_Certifying`

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

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 $i_W : \overline{W} \to W$. The checker program C receives a triple $(\overline{x}, \overline{y}, \overline{w})$ and is supposed to check whether it fulfills 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 (e.g., run forever or halt with an arbitrary output). If $\varphi(x)$, C must halt and either accept or reject. A correct checker C will accept if $\mathcal{W}(x, y, w)$ holds and reject otherwise. The following proof obligations arise:

Witness Property: A proof for the implication (1).
Checker Correctness: A proof that C checks the witness predicate if the precondition φ is satisfied. I.e., for an input $(\overline{x}, \overline{y}, \overline{w})$ with $x = i_X(\overline{x})$, $y = i_Y(\overline{y})$, $w = i_W(\overline{w})$:
1. If $\varphi(x)$, C halts.
2. If $\varphi(x)$, C accepts if and only if $\mathcal{W}(x, y, w)$.

Tools Isabelle/HOL [23] is an interactive theorem prover for classical higher-order logic based on Church's simply-typed lambda calculus. The system is built on top of a kernel providing a small number of inference rules; complex deductions (especially by automatic proof methods) ultimately rely on these rules only. This strategy [14] guarantees correctness as long as the inference kernel is correct. Isabelle/HOL comes with a rich set of already formalized theories, e.g., natural numbers, integers, sets, finite sets, and as a recent addition graphs [24]. Proofs in Isabelle/HOL can be 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.

Simpl [25] is a generic imperative language designed to allow a deep embedding of real programming languages such as C into Isabelle/HOL for the purpose of program verification. The C-to-Isabelle parser converts a large subset of C99-code into low-level Simpl code. Simpl provides the usual imperative language constructs such as functions, variable assignments, sequential composition, conditional statements, while loops, and exceptions. There is no return statement for abrupt termination; it is emulated by exceptions. Simpl has no expression language of its own; rather, every Isabelle expression is also a Simpl expression. Programs may be annotated by invariants. Specifications for Simpl programs are given as Hoare triples, where pre- and post-condition are arbitrary Isabelle expressions. A verification condition generator (*VCG*) converts Hoare Triples to a set of higher-order formulas.

The C-to-Isabelle parser makes no effort to abstract from details of the C-language. AutoCorres [15] builds upon this parser and, in a fully verified way, provides a simpler representation of the original program. Apart from simplifying the control flow, it transforms the deeply embedded Simpl code into a shallowly embedded monadic representation where local variables are modeled as bound Isabelle variables. There are multiple monads from which AutoCorres chooses depending on the C features used; the most common one is the nondeterministic state monad. In this monad, program statements are a function from a heap to a tuple consisting of a failure flag and the non-deterministic state, represented as a set of pairs of return value and heap. The monadic bind operation implements sequential composition. Again, specifications are given as Hoare triples and a VCG converts these to higher-order formulas [11].

VCC [12] is an assertional, automatic, deductive code verifier for full C code. Source code is annotated with specifications in the form of function contracts, data invariants, loop invariants, and further annotations to guide the verifier. Annotated code can still be compiled with a normal C compiler. From the annotated program, VCC generates verification conditions for partial or total correctness, which it then tries to discharge automatically.

3 Verification of Checkers within Isabelle/HOL

The VCC approach was used to verify several checkers in the field of graphs from the algorithmic library LEDA. For the sake of comparison, we rework the verification of the connectedness checker. Moreover, we verify the LEDA checker for testing graph non-planarity. In order to get a measure of the effort dedicated to the verification of the algorithm respectively that of dealing with C-intricacies, we use two methods to verify the checker in Isabelle/HOL: First, we verify an implementation in Simpl. Second, we use AutoCorres to verify a C implementation. We compare the approaches in Section 5.

Connectedness of Graphs Given an undirected graph $G = (V, E)$, we consider an algorithm that decides whether G is connected, i.e., whether there is a path between any pair of vertices [21, Section 7.4]. Non-connectedness is certified by a cut, i.e., a nonempty subset S of the vertices with $S \neq V$, such that every edge of the graph has either both or no endpoint in S. Connectedness is certified by a spanning tree of G. On a high level, we instantiate the general framework as follows:

input x	=	an undirected graph $G = (V, E)$
output y	=	either True or False indicating whether G is connected
witness w	=	a cut or a spanning tree
$\varphi(x)$	=	G is well-formed, i.e., $E \subseteq V \times V$, V and E are finite sets
$\mathcal{W}(x, y, w)$	=	y is True and w is a spanning tree or y is False and w is a cut
$\psi(x, y)$	=	if y is True, G is connected and if y is False, G is not connected.

As in previous work [2], we restrict ourselves to the positive case $y = \mathit{True}$. For an example of a graph and its witnessing spanning tree see Fig. 3 in [2]. We represent spanning trees by functions $\mathit{parent\text{-}edge}$ and num and a root vertex r and view the edges of the tree oriented towards r: for each v, $\mathit{parent\text{-}edge}(v)$ is the first edge on the path from v to r (we set $\mathit{parent\text{-}edge}(r) = \mathit{None}$), and $\mathit{num}(v)$ is the length of this path. Undirected graphs are represented as bidirected graphs, i.e., for every unordered edge $\{u, v\}$ of G, we have ordered pairs (u, v) and (v, u) in the representation of G.

Witness Property The witness property states that if the conditions in Fig. 1 hold, the graph is connected. This was already proven in Isabelle/HOL by Alkassar et al. [2]. We extend the theorem to also state that the conditions imply that the num-value of each vertex is its depth in the spanning tree. This is important for the C-verification.

locale *connected-components-locale* = *pseudo-digraph* +
  fixes $num : \alpha \Rightarrow nat$ **and** $parent\text{-}edge : \alpha \Rightarrow \beta\ option$ **and** $r : \alpha$
  assumes $r\text{-}assms$: $r \in verts\ G \wedge parent\text{-}edge\ r = None \wedge num\ r = 0$
  assumes $parent\text{-}num\text{-}assms$: $\bigwedge v.\ v \in verts\ G \wedge v \neq r \Longrightarrow$
    $\exists\ e \in \text{arcs}\ G.\ parent\text{-}edge\ v = Some\ e \wedge head\ G\ e = v \wedge num\ v = num\ (tail\ G\ e) + 1$

Fig. 1: Preconditions for the connectedness proof in Isabelle. G is a well-formed graph with vertices of type α and edges of type β.

Simpl Implementation and Verification We represent graphs as in previous work [2]. The type $IGraph$ represents a graph G by the numbers $ivertex\text{-}cnt\,G$ and $iedge\text{-}cnt\,G$ of its vertices and edges and a function $iedges\,G$, mapping $0 \leq i < iedge\text{-}cnt\,G$ to the pair of endpoints of the i-th edge. A graph is *well-formed* if all endpoints are smaller than $ivertex\text{-}cnt\,G$.

Each of the conditions in Fig. 1 is checked by a procedure. For example, the procedure $parent\text{-}num\text{-}assms$ in Fig. 2 checks $parent\text{-}num\text{-}assms$ in the obvious way. The loop invariant $parent\text{-}num\text{-}assms\text{-}inv$ states that $parent\text{-}num\text{-}assms$ holds up to vertex i. ***VAR MEASURE*** introduces the measure function used for the termination proof and the command ***ANNO*** binds logical variables to be used in the invariant. Total correctness of each function is formulated as a Hoare triple; see Lemma $parent\text{-}num\text{-}assms\text{-}spec$ in Fig. 2. Invoking the VCG and using the annotations (loop invariant and measure function) is sufficient for the correctness proof.

C Implementation and Verification The C representation of graphs is similar to that in Simpl. In particular, numbers are now of bounded precision. This means we need to prove absence of overflows during verification. The number of vertices and edges are now ***unsigned int***s. We represent spanning trees as explained above, but use arrays instead of functions. The function $parent\text{-}edge$ is represented as an array of (signed) ***int***, and num as an array of ***unsigned int***. As in previous work [2], we require as a precondition that the input graph is well-formed.

The $check\text{-}connected$ checker is a function that accepts exactly when the two functions $check\text{-}r$ and $check\text{-}parent\text{-}num$ accept. 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 $parent\text{-}edge[v]$ is incident to v and that the other endpoint of the edge has a number one smaller than $num[v]$.

The first step in the C verification is calling the C-to-Isabelle parser and invoking AutoCorres. As in Simpl, for each function in the code we prove a corresponding specification lemma, formulated as a Hoare triple and reasoned about using a VCG. The termination proof of the checkers is as trivial as in the Simpl case. For proving functional correctness, we introduce some helper functions that assist in relating the implementation types to Isabelle types. For example, the abstraction predicate array list, $arrlist$, takes as input the state of the heap h, a list l and a pointer p and checks whether p points in h to an array containing the values of l. We also introduce a set of lemmas to ease dealing with bounded numbers.

We prove that the checker function checks the conditions in Fig. 1. This proof happens under the assumption that the pointers to the graph, to its edges, to num and to $parent\text{-}edge$ can be abstracted to Isabelle datatypes (using the $arrlist$ predicate).

```
definition $\mathit{parent\text{-}num\text{-}assms\text{-}inv} : \mathit{IGraph} \Rightarrow \mathit{IVert} \Rightarrow \mathit{IPEdge} \Rightarrow \mathit{INum} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$
where $\mathit{parent\text{-}num\text{-}assms\text{-}inv}\ G\ r\ p\ n\ k \equiv \forall\ k < i.\ i \neq r \rightarrow (\textbf{case}\ p\ i\ \text{of}\ \mathit{None} \Rightarrow \mathit{False}$
  $\mid \mathit{Some}\ x \Rightarrow x < \mathit{iedge\text{-}cnt}\ G \wedge \mathit{snd}\ (\mathit{iedges}\ G\ x) = i \wedge n\ i = n\ (\mathit{fst}\ (\mathit{iedges}\ G\ x)) + 1)$

procedures $\mathit{parent\text{-}num\text{-}assms}$
  (G : $\mathit{IGraph}$, r : $\mathit{IVert}$, parent-edge : $\mathit{IPEdge}$, num : $\mathit{INum}$ | R : $\mathit{bool}$)
in ANNO $(G, r, p, n)$. $\{\!|$ G = $G$ $\wedge$ r = $r$ $\wedge$ parent-edge = $p$ $\wedge$ num = $n$ $|\!\}$
 where vertex : $\mathit{IVert}$, edge-id : $\mathit{Edge\text{-}Id}$
   R := $\mathit{True}$ ; vertex := 0 ;
  TRY
   WHILE vertex < $\mathit{ivertex\text{-}cnt}$ G
   INV $\{\!|$ R = $\mathit{parent\text{-}num\text{-}assms\text{-}inv}$ G r parent-edge num vertex
     $\wedge$ vertex $\leq$ $\mathit{ivertex\text{-}cnt}$ G$|\!\}$ VAR MEASURE ($\mathit{ivertex\text{-}cnt}$ G $-$ vertex)
   DO
    IF (vertex $\neq$ r) THEN
     IF parent-edge vertex = $\mathit{None}$ THEN R := $\mathit{False}$ ; THROW FI ;
     edge-id := the (parent-edge vertex) ;
     IF edge-id $\geq$ $\mathit{iedge\text{-}cnt}$ G $\vee$ $\mathit{snd}$ ($\mathit{iedges}$ G edge-id) $\neq$ vertex
       $\vee$ num vertex $\neq$ num ($\mathit{fst}$ ($\mathit{iedges}$ G edge-id)) + 1 THEN R := $\mathit{False}$ ; THROW FI
    FI ;
    vertex := vertex + 1
   OD
  CATCH SKIP END $\{\!|$R=$\mathit{parent\text{-}num\text{-}assms\text{-}inv}$ G r parent-edge num ($\mathit{ivertex\text{-}cnt}$ G)$|\!\}$

lemma (in $\mathit{parent\text{-}num\text{-}assms\text{-}impl}$) $\mathit{parent\text{-}num\text{-}assms\text{-}spec}$:
 $\forall G\ r\ p\ n.\ \Gamma \vdash_t$$\{\!|$G = $G$ $\wedge$ r = $r$ $\wedge$ parent-edge = $p$ $\wedge$ num = $n$$|\!\}$
  R := PROC $\mathit{parent\text{-}num\text{-}assms}$(G, r, parent-edge, num)
  $\{\!|$ R = $\mathit{parent\text{-}num\text{-}assms\text{-}inv}\ G\ r\ p\ n\ (\mathit{ivertex\text{-}cnt}\ G)$$|\!\}$
```

Fig. 2: Excerpts from the Simpl implementation and verification of connectedness. The Lemma *parent-num-assms-spec*, formulated as a Hoare triple, states that the procedure *parent-num-assms* terminates (indicated by $\vdash_t$) and computes *parent-num-assms-inv*. Observe the distinction between logical and program variables; x versus x for a variable with name x.

Experiences and Lessons Learned The verification of this checker assures us that the AutoCorres approach is feasible. The effort for the verification of the C-version of the connectedness checker was about the same as in the VCC approach. VCC knows more about C and this made it easier to reason about the C-program. This advantage would show even more clearly in programs that use low-level features of C more intensively, e.g., bit operations on words. On the other hand, one is forced to formalize a small number of graph-theoretic concepts such as path in two logical systems, this complicates the VCC-approach. A small number sufficed because verifying that the C-checker correctly checks the assumptions from Fig. 1 needs no graph-theoretic knowledge and hence there is a clear separation of labor between VCC and Isabelle/HOL. The disadvantage of double formalization shows more clearly in programs that need complex mathematical reasoning in the checker correctness proof and hence would require formalizing more advanced concepts in VCC. The checker for non-planarity presented in the next section is an example to this effect. There the correctness proof of the program requires graph-theoretic reasoning. If we had tried to verify this example using the VCC-approach, we would have had to formalize a non-trivial theory twice.

The connectedness checker verified using the VCC approach [2] has an unintended weakness. Not every representable connected graph has a spanning tree that could be represented as input to the checker. This is because the vertices of the graph were represented as ***unsigned int*** and the array num had type ***unsigned short***; this holds true for the program actually verified, not for the program listed in the paper. Thus graphs having no spanning tree of depth bounded by the size of ***unsigned short*** had no representable witness. VCC had no difficulties in automatically verifying that the addition in the C equivalent of num *(fst (iedges* G edge-id*)) + 1* (see Fig. 2) does not overflow, because types smaller than ***int*** are lifted to ***int*** for arithmetic operations in C. In the AutoCorres verification, we had to manually prove that $s + 1 \leq u$, where s and u are the maximum values of ***unsigned short*** and ***int***, respectively. This led us to notice and modify the type of num in the checker to ***unsigned int***. Now the addition could potentially overflow and we need to show that it does not. This is proven by strengthening the loop invariant to infer that num-value cannot exceed the number of vertices and hence does not overflow in a correct witness. In order to prove that the checker accepts if and only if the assumptions in Listing 1 hold one needs the stronger witness property mentioned above. Even though in this case manually discharging guards was useful, it demonstrates that VCC saves effort when it comes to automatically discharging guards.

Non-Planarity of Graphs One of the motivating examples for the introduction of certified algorithms in the LEDA library is the planarity test [21]. The planarity check in LEDA takes as input a graph x and returns $y = \mathit{True}$ and a combinatorial planar embedding w of x if x is planar or $y = \mathit{False}$ and a Kuratowski subgraph w of x otherwise. On a high level, we instantiate the general framework as follows:

input x	=	an undirected graph $G = (V, E)$, possibly with loops
output y	=	either True or False
witness w	=	combinatorial planar embedding or Kuratowski subgraph
$\varphi(x)$	=	G is well-formed, i.e., $E \subseteq V \times V$ where V and E are finite.
$\psi(x, y)$	=	If y is True, x is planar, else x is not planar.

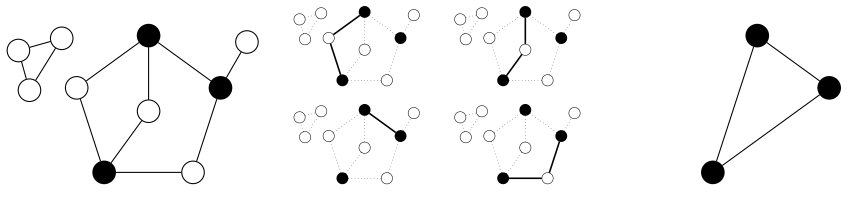
In this paper, we restrict ourselves to the case $y = \mathit{False}$. Then $\mathcal{W}(x, \mathit{False}, w)$ holds iff w is a Kuratowski subgraph of x. Let K_5 be the complete graph on five vertices and $K_{3,3}$ the complete bipartite graph on three and three vertices. We call $K_{3,3}$ and K_5 *Kuratowski graphs*. Kuratowki's theorem is the basis for our formalization of non-planarity (see Fig. 3).

Theorem 1 (Kuratowski). *A graph K is a* Kuratowski subgraph *of G if K is a subgraph of G and the subdivision of a Kuratowski graph. A graph G is planar if and only if it has no Kuratowski subgraph.*

Witness predicate The key step of the checker is testing whether the certificate K is a subdivision of a $K_{3,3}$ or K_5. One option is to repeatedly take a node of degree 2

$$subdivide(K, (u, v), w) = (V(K) \cup \{w\}, (E(K) \setminus \{uv\}) \cup \{uw, vw\})$$
$$planar(G) = \neg(\exists K.\ K \leq G \wedge (\exists H.\ subdivision(H, K) \wedge (K_{3,3}(H) \vee K_5(H))))$$

Fig. 3: Characterization of planarity. $subdivision(H, K)$ is the minimal predicate satisfying the following rules: H is a subdivision of itself and if K is a subdivision of H, e is an edge of K, and w is a new vertex, then $subdivide(K, e, w)$ is a subdivision of H. By $\leq$, we denote the subgraph relation.



(a) G (b) all ipaths (c) contracted graph

Fig. 4: A graph G and its ipaths and contracted graph ($V_3(G)$ in black). Neither the isolated circle nor the node of degree 1 are on any ipath (or in $V_3(G)$), so they do not contribute to the contracted graph.

and contract it. In an imperative implementation this requires the program to work on a copy of K (or to modify the input). Instead, we follow the method used in LEDA [21] and compute the *contraction* of K in single step and check whether the contraction is a Kuratowski graph. This requires only a constant amount of memory.

Definition 1 (Contraction). *Let G be a graph and $V_3(G)$ be the set of all vertices of G with degree at least three. Let E' be such that $uv \in E'$ iff $u, v \in V_3(G)$ and there is a path in G connecting u and v whose interior vertices are not in $V_3(G)$. Then $G' = (V_3(G), E')$ is the* contraction *of G. A path with end nodes in $V_3(G)$ and interior nodes in $V(G) \setminus V_3(G)$ is called an* ipath*. See Fig. 4 for an illustration.*

Note that in general G is not a subdivision of its contraction. In particular, vertices of degree one or less and isolated cycles are discarded and cannot be reconstructed by subdivision. Nevertheless, contraction gives us a useful over-approximation of the Kuratowski subgraphs, as demonstrated by the following lemmas.

Lemma 1. *Let K be a graph and H the contracted graph of K. Then there exists a subgraph K' of K such that K' is a subdivision of H. In particular, if H is a $K_{3,3}$ or K_5 and K a subgraph of a graph G, then G is not planar.*

Lemma 2. *Let H be a Kuratowski graph. If K is a subdivision of H, then H is the contracted graph of K. In particular, if K is a Kuratowski subgraph of a graph G, then the contracted graph of K is a Kuratowski graph.*

We prove both properties in Isabelle. To this end, we introduce the class of slim graphs. These correspond to those graphs on which contraction is an inverse to subdivision. The contraction of a non-slim graph G is also the contraction of a slim subgraph of G and the above lemmas derive from that. For details of the proof see [24].

Based on this, we give a new witness predicate $\mathcal{W}'$ as follows: $\mathcal{W}'(x, \mathit{False}, w)$ holds if and only if w is well-formed and a loop-free subgraph of x such that the contracted graph of w is a Kuratowski graph. Then Lemma 1 ensures the witness property. Lemma 2 ensures that $\mathcal{W} \subseteq \mathcal{W}'$, i.e., all certificates of non-planarity are accepted.

Implementation and Verification The implementation of the checker is roughly divided into four steps: (1) Test whether K is a subgraph of G. (2) Test whether K is loop-free. (3) Compute H by contracting K. (4) Test whether H is a Kuratowski graph. The input is accepted if and only if all four tests succeed. The test for loop-freeness is not needed for correctness, but simplifies the verification of the contraction step. We verified the full algorithm, but focus on step (3) in this write-up. We use a different representation of graphs than in the previous example (see Fig. 5), as we need to encode vertices explicitly (and not only the number of vertices) to represent subgraphs.

The code to compute the contraction of K consists of three parts: First, the graph H is created by taking all vertices of degree three or more (and no edges) of K; if there are more than 6 such vertices, the certificate is rejected. The core of the computation is then performed by the function *find-endpoint* (Fig. 6): For a given vertex $v_{start} \in V(H)$ and an incident edge $e \in E(G)$ (given by its other endpoint v_{next}), it computes implicitly the ipath of G starting with this edge end returns its last vertex (if it exists). The contracted edge described by this ipath is then added to H.

Checker Correctness We assume that the input and certificate are well-formed graphs. Most of the termination arguments are pretty trivial (loops counting upwards to some constant), but termination of *find-endpoint* is not obvious: The procedure implicitly constructs an ipath, adding a vertex in every iteration. Termination follows as the length of an ipath is bounded by the number of vertices.

For partial correctness, the checker returns true if and only if $\mathcal{W}'(x, False, y)$ holds. In the verification, most of the work is needed for step (3). To prove the specification of *find-endpoint* (Fig. 7) one needs to show that a maximal path where all interior nodes are of degree two is uniquely determined by its first edge. From this it follows relatively easily that calling *find-endpoint* for all nodes and their incident edges determines all edges of the contracted graph. Without referring to the mathematical background theory, both termination and partial correctness would be hard to prove.

Verifying the C implementation There are some differences between the Simpl and C implementations. In C, Graphs are not represented as a pair of lists, but as a struct with two pointers to arrays, and instead of natural numbers, we use bounded machine words. Finally, in Simpl, basic graph operations like “vertex contained” were stated as Isabelle expressions. In C, they need to be implemented and verified.

AutoCorres provides a natural translation of C code, so we hoped that for the verification of the C program, we could start with the the Simpl proof and fill in the gaps: i.e., abstract memory accesses and datatypes to the ones used in the Simpl proof and verify the functions not implemented before. The latter was indeed straight-forward. Similarly, abstracting the heap to the graph datatypes of Isabelle was tedious, but straight-forward, following established schemes [22]. Most of the additional effort was needed because of

```
struct edge_t {                 struct graph_t {                   struct contr_t {
    unsigned start;                 unsigned vert_cnt;                 unsigned char vert_cnt;
    unsigned target; };             unsigned edge_cnt;                 unsigned verts[6];
                                    unsigned *verts;                   unsigned char
                                    struct edge_t *edges; };             edges[6][6]; };
```

Fig. 5: C datastructures for graphs. *graph_t* represents a graph by a list of vertices and a list of edges. *contr_t* represents the contracted graph as an adjacency matrix.

```
procedures find-endpoint (G : IGraph',
  H : IGraph', v_start : IVert, v_next : IVert
  | R : IVert option)
where
 found : bool, i : nat, len : nat, v_0 : IVert,
 v_1 : IVert, v_t : IVert io-edges : ig-edge list,
TRY
 IF v_start = v_next THEN RAISE R := None FI ;
 v_0 := v_start ; v_1 := v_next ; len := 1 ;
 WHILE v_1 ∉ set (ig-verts H) DO
  io-edges := ig-in-out-edges G v_1 ;
  i := 0 ; found := False ;
  WHILE ¬found ∧ i < length io-edges DO
   v_t := ig-opposite G (io-edges ! i) v_1 ;
   IF v_t ≠ v_0 THEN
    found := True ; v_0 := v_1 ; v_1 := v_t FI ;
   i := i + 1
  OD ;
  len := len + 1 ;
  IF ¬ found THEN RAISE R := None FI
 OD ;
 IF v_1 = v_start THEN RAISE R := None FI ;
 R := Some v_1
CATCH SKIP END
```

```
unsigned find_endpoint(struct graph_t *g,
  struct contr_t *h, unsigned v_start,
  unsigned v_next) {
    unsigned v0 = v_start;
    unsigned v1 = v_next;

    while (tmp_get_index(h, v1) ≡ −1) {
        unsigned i;
        for (i=0; i < edge_cnt(g); i++) {
            unsigned vt = opposite(v1,
              edge(g,i));
            if (vt ≠ v0 ∧ vt ≠ −1) {
                v0 = v1;
                v1 = vt;
                break;
            }
        }
        if (i ≡ edge_cnt(g)) return −1;
    }
    if (v1 ≡ v_start) return −1;
    return v1;
}
```

Fig. 6: The function *find-endpoint* in Simpl and C. H (resp. h) is the preliminary contracted graph. The function implicitly constructs an ipath by adding vertices until a vertex of degree 3 (i.e., in H) is reached. The if-statement in the inner loop ensures that the algorithm does not go back the edge from the previous iteration. If the outer loop aborts abnormally, then no vertex in H is reachable from $\mathsf{v}_{\mathsf{start}}$ via $(\mathsf{v}_{\mathsf{start}}, \mathsf{v}_{\mathsf{next}})$. The Simpl implementation uses relatively high-level datastructures, like sets and list.

$$\begin{array}{l}\forall\sigma.\ \Gamma \vdash_t \{\!|\sigma.\ \mathit{iverts}\ \mathsf{H} = \mathit{iverts}_3\ \mathsf{G} \wedge \textit{loop-free}\ (\textit{mk-graph}\ \mathsf{G}) \wedge \mathsf{v}_{\mathsf{start}} \in \mathit{set}\ (\mathit{iverts}\ \mathsf{H}) \\ \quad \wedge\ \mathit{iadj}\ \mathsf{G}\ \mathsf{v}_{\mathsf{start}}\ \mathsf{v}_{\mathsf{next}} \wedge \textit{IGraph-inv}\ \mathsf{G}|\!\} \\ \quad \mathsf{R} := \mathbf{PROC}\ \textit{find-endpoint}(\mathsf{G}, \mathsf{H}, \mathsf{v}_{\mathsf{start}}, \mathsf{v}_{\mathsf{next}}) \\ \{\!|\mathbf{case}\ \mathsf{R}\ \text{of}\ \mathit{None} \Rightarrow \neg(\exists \mathrm{p}\ \mathrm{w}.\ \mathit{ipath}\ (\textit{mk-graph}\ {}^{\sigma}\mathsf{G})\ {}^{\sigma}\mathsf{v}_{\mathsf{start}}\ ({}^{\sigma}\mathsf{v}_{\mathsf{start}}{}^{\sigma}\mathsf{v}_{\mathsf{next}}\ \#\ \mathrm{p})\ \mathrm{w}) \\ \qquad\qquad |\ \mathit{Some}\ \mathrm{w} \Rightarrow (\exists \mathrm{p}.\ \mathit{ipath}\ (\textit{mk-graph}\ {}^{\sigma}\mathsf{G})\ {}^{\sigma}\mathsf{v}_{\mathsf{start}}\ ({}^{\sigma}\mathsf{v}_{\mathsf{start}}{}^{\sigma}\mathsf{v}_{\mathsf{next}}\ \#\ \mathrm{p})\ \mathrm{w})\ |\!\}\end{array}$$

Fig. 7: Specification of *find-endpoint*: If H has all degree-3 nodes of G and G has no loops, then the procedure decides the existence of an ipath starting with the nodes $\mathsf{v}_{\mathsf{start}}$ and $\mathsf{v}_{\mathsf{next}}$. *mk-graph* abstracts a graph and ${}^{\sigma}\mathsf{x}$ refers to the value of x before the execution.

the bounded precision integers. This was somewhat surprising, because the only arithmetic operations occurring in the program are equality and increment against a fixed upper bound.

There are mainly two reasons for the problems we encountered with words: First, Isabelle has only weak support for proving properties involving words automatically. Second, such properties often occur not on their own, but as side-conditions in a larger proof. While Isabelle's automatic proof tools can often discharge such properties for natural numbers, they cannot do so for words and therefore fail, leaving the user to solve the goal mostly manually.

4 Abstraction

The issues with reasoning about functions using words motivated us to implement an abstraction framework for AutoCorres programs. The idea is to take the original function f and give a modified implementation f' that uses natural numbers instead of words. With the help of the abstraction framework, we prove f and f' to be equivalent and then perform verification on the abstracted function.

Abstraction or *refinement* is a well-known idea going back to Dijsktra [13] and Wirth [28] and put into a formal calculus by Back [3]. In particular, AutoCorres uses this technique to get from the Simpl program generated by the C parser to the simplified version presented to the user. We want to change two things in the abstraction process: Where appropriate, we want to replace words by natural numbers. For this, we also need to insert a guard before each operation on words that asserts that there will be no overflow (these guards then need to be discharged in the correctness proof of the abstract function). Moreover, we want to be able to insert ghost code (and ghost state), i.e., additional computations which have no influence on the outcome of the function. Such ghost code is often useful for stating loop invariants.

However, the abstraction framework used by AutoCorres [27] is unsuited for our purposes: It expects that each state in the concrete program corresponds to at most one state in the abstract program. This makes it difficult to insert ghost code. Moreover, although the given proof rules are syntax directed they must be applied in a guided manner, this makes them harder to use. Therefore, we give our own definition.

Recall that a computation in the nondeterministic state monad returns a failure flag and a set of states. For a relation rel on states, we define a relation $refines$ on abstract. resp. concrete program statements A and C:

$$\mathit{refines}\ \mathit{rel}\ A\ C = (\neg \mathit{fail}\ A \longrightarrow (\neg \mathit{fail}\ C \wedge \forall c \in \mathit{st}\ C.\ \exists a \in \mathit{st}\ A.\ (a, c) \in \mathit{rel})$$

In particular, instead of proving correctness for the concrete program, we can prove correctness for the abstract program. We only give a simplified version here, which does not allow abstracting the heap. A state is a pair (r, h), where r is the return value of the previous command (often a tuple) and h is the heap.

Lemma 3. *Let P_A and P_C be programs (i.e., functions from state to program state) and rel be a relation on states. The Hoare triple*

$$\{\!|Q|\!\}\ P_C\ \{\!|R\ |\!\}!$$

states that P_C is totally correct w.r.t. to the precondition Q (a predicate on heaps) and the postcondition R (a predicate on value/heap pairs). Assume that

$$\forall h.\ Q\ h \longrightarrow \mathit{refines}\ \mathit{rel}\ (P_A\ h)\ (P_C\ h)$$

i.e., for all heaps satisfying a precondition Q the result of the abstract and concrete programs are related. In addition, assume that

$$\{\!|Q|\!\} P_A \{\!|\lambda r_A\ s_A.\ \forall r_C\ s_C.\ ((r_A, s_A), (r_C, s_C)) \in \mathit{rel} \longrightarrow R\ r_C\ s_C|\!\}!$$

i.e., for a heap h satisfying Q, the result of $P_A\ h$ is related to a concrete program state satisfying R. Then the concrete program P_C satisfies the specification above.

To prove that two programs are related, we provide a syntax directed proof procedure which compares the two programs instruction by instruction. This requires the two programs to be very similar in structure. This is the case in our application. The central rule of the proof procedure is the rule for sequential composition.

Lemma 4 (Refinement of Sequential Composition). *Let $P_{A,1}$ be a program and $P_{A,2}$ a function mapping a return value of $P_{A,1}$ to a program (similarly for $P_{C,1}$, $P_{C,2}$).*

$$\begin{aligned}&[\![\mathit{refines}\ \mathit{rel}\ (P_{A,1}\ h_A)\ (P_{C,1}\ h_C);\\ &\ \forall ((r_A, h_A), (r_C, h_C)) \in \mathit{rel}.\ \mathit{refines}\ \mathit{rel}'\ (P_{A,2}\ r_A\ h_A)\ (P_{C,2}\ r_C\ h_C)]\!]\\ &\quad \Longrightarrow \mathit{refines}\ \mathit{rel}'\ ((P_{A,1} \ggg P_{A,2})\ h_A)\ ((P_{C,1} \ggg P_{C,2})\ h_C)\end{aligned}$$

Here, $\ggg$ is the operator for sequential composition. $(P_1 \ggg P_2)\ h$ calls P_1 with heap h, and, for every pair (r_2, h_2) in the result of $P_1\ h$, calls $P_2\ r_2\ h_2$. The union of the results is returned.

Note that the relation rel w.r.t. which $P_{A,1}$ and $P_{C,1}$ are refined is not fixed a priori. Our verification condition generator will synthesize it during the proof, using the following basic blocks:

- A refinement relation for words: $\{(n, w) \mid n = \mathit{unat}\ w\}$. Here, *unat* is the conversion from words to natural numbers.
- The identity relation.
- A ghost relation, allowing the introduction of an additional stack variable in the abstracted program: $\{(((g, r), h), (r', h')) \mid ((r, h), (r', h'))\}$

These are put together with the help of a pairing relation $\{((r, h), (r', h')) \mid (r, r') \in \mathit{rrel} \wedge (h, h') \in \mathit{hrel}\}$.

Putting abstraction to use For the Kuratowski checker, our proof process is as follows: For each function f containing word arithmetic, we make a copy f' of this function, in which we replace words by natural numbers. For each arithmetic operation, we insert a guard stating that this operation would not overflow on words (see Fig. 8). Where necessary, we also add ghost code and annotate loops with invariants. One example of this is function *find-endpoint*, where we add an variable holding the computed ipath and use this in the invariant. Note that the ghost code can use arbitrary Isabelle expressions. Then we prove that f' is an abstraction of f, using the verification condition generator sketched in the previous section. The proof is mostly automatic; we only need to prove simple properties about words and natural numbers.

return $((i : 32\ word) + 1)$

guard $(\lambda\_. (i : nat) < unat\ (max\text{-}word : 32\ word))$;
return $(i + 1)$

(a) concrete program (b) abstract program

Fig. 8: Abstraction of word arithmetic. The guard ensures that the operations on words and natural numbers behave the same. For the refinement proof, we write *ADD_guard* instead of *guard* as a hint for our syntax directed VCG.

5 Evaluation

After abstraction, verification of the non-planarity checker follows closely the proof of the Simpl program. Overall, we conclude that the use of AutoCorres provides a viable alternative to the VCC approach for the verification of certifying computations. Moreover, we can profit from a previous verification of the algorithm. However, it is necessary to lift the C program to a similar level of abstraction as the pseudo code. This could not be achieved with the facilities provided by AutoCorres alone, but required us to implement our own refinement framework. The effort of developing this framework is required only once and can be reused for future verifications.

It is worth noting that there is parallel work adding automatic abstraction of words into AutoCorres [16]. However, when verifying a program, one is likely to encounter other datatypes that need a custom abstraction. In addition, our abstraction framework gives the option of adding ghost code, which is known to ease the formulation of invariants.

Both the Simpl and the C implementation of the Kuratowski checker consist of around 300 lines of code (the Simpl syntax is more verbose than C). The verification of the Simpl checker was done in 1300 lines. The verification of the C checker required 3200 lines and 1400 lines for the refinement framework. Of the 3200 lines, 900 deal with heap abstraction and access and the verification of basic graph operations not implemented in the Simpl code.

6 Related Work

Verifying code within interactive theorem provers is a an active field of research. The seL4 microkernel that is written in low-level C was verified within Isabelle/HOL using the C-to-Isabelle parser [17]. The underlying approach is refinement starting from an abstract specification via an intermediate implementation in Haskell to the final C code. Coq [5] was used both for programming the CompCert compiler and for proving its correctness [18]. CFML is a verification tool embedded in Coq that targets imperative Caml programs [10]. It was used to verify several imperative data structures.

Shortest path algorithms, especially imperative implementations thereof, are popular as case studies for demonstrating code verification [10, 8]. They target full functional correctness as opposed to instance correctness. Verifying instance correctness is orthogonal to verifying the implementation of a particular algorithm and it is a tempting choice that also attracted much attention. In 1997, a checker for sorting algorithms has been developed and verified [9]. The DeCert project aims to design an architecture where either decision procedures are proven correct within Coq or produce witnesses allowing external checkers to verify the validity of their results, [6] provides an example. In recent work [2], a general framework to verify certifying computations is developed.

7 Conclusion

In this paper, we explored an alternative to the VCC approach, which provides higher trust guarantees, and verified checker for graph non-planarity. To our knowledge, no algorithm or checker for graph non-planarity was verified before.

The LEDA project [21] has shown that the concept of certifying computations eases the construction of libraries of reliable implementations of complex combinatorial and geometric algorithms. Reliability is increased because the output of every computation is checked for correctness by a checker program. Checker programs are relatively simple and hence easier to implement correctly than the corresponding solution algorithms. Certifying algorithms are available for a large number of algorithmic problems [19].

Our AutoCorres approach does not use VCC; the entire verification is done in Isabelle/HOL. We did so for three reasons: (1) The VCC approach, with its use of two different tools requires the formalization of certain concepts in two theories, a duplication of effort. (2) Furthermore, it requires trust in VCC, a fairly complex program. We have no reason not to trust the program. However, as a matter of principle, the trusted code base should be kept as small and simple as possible. (3) The recent tool AutoCorres [15] promised to greatly simplify reasoning about C in Isabelle.

Our experience with AutoCorres is positive. The AutoCorres approach presented in this paper yields a viable alternative to the VCC approach. It is particularly useful when the verification requires domain-specific reasoning (e.g., graph theory, as it was the case for the non-planarity checker).

The implementation of each of the advanced algorithms in LEDA took several man-months (recollection of the third author). In comparison, with either approach, it took less time to verify the checker. Note that the non-planarity checker is amongst the most complex checkers in LEDA. The verification time is likely to go down with increased experience and development of the tools (cf. [16]). In particular, we extended AutoCorres with a reusable abstraction framework. We find that our work demonstrates that the development of libraries of certifying programs with formally verified checkers is feasible at reasonable cost.

Acknowledgement We thank David Greenaway and Thomas Sewell for their advice on using AutoCorres and for their feedback on the paper. We also thank Jasmin Christian Blanchette for his feedback on the paper.

References

1. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows. Prentice Hall (1993)
2. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. JAR (2013). DOI 10.1007/s10817-013-9289-2
3. Back, R.J.R.: Correctness preserving program refinements: Proof theory and applications. Mathematical Centre tracts. Mathematisch centrum (1980)

4. de Berg, M., Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Springer (1997)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
6. Besson, F., Jensen, T.P., Pichardie, D., Turpin, T.: Certified result checking for polyhedral analysis of bytecode programs. In: TGC, pp. 253–267 (2010)
7. Blum, M., Kannan, S.: Designing programs that check their work. In: STOC, pp. 86–97 (1989)
8. Böhme, S., Leino, K.R.M., Wolff, B.: HOL-Boogie—An interactive prover for the Boogie program-verifier. In: TPHOLs, *LNCS*, vol. 5170, pp. 150–166 (2008)
9. Bright, J.D., Sullivan, G.F., Masson, G.M.: A formally verified sorting certifier. IEEE Transactions on Computers **46**(12), 1304–1312 (1997)
10. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP, pp. 418–430 (2011)
11. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: TPHOLs, *LNCS*, vol. 5170, pp. 167–182 (2008)
12. 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: TPHOLs, *LNCS*, vol. 5674, pp. 23–42 (2009)
13. Dijkstra, E.W.: Notes on structured programming. Technological University Eindhoven Netherlands (1970)
14. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, *LNCS*, vol. 78 (1979)
15. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Interactive Theorem Proving, *LNCS*, vol. 7406, pp. 99–115 (2012)
16. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of c code without the pain. In: PLDI, p. To appear (2014)
17. 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. CACM **53**(6), 107–115 (2010)
18. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7), 107–115 (2009)
19. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011)
20. Mehlhorn, K., Näher, S.: From algorithms to working programs: On the use of program checking in LEDA. In: MFCS, *LNCS*, vol. 1450, pp. 84–93 (1998)
21. Mehlhorn, K., Näher, S.: The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
22. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Information and Computation **199**, 200–227 (2005)
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283 (2002)
24. Noschinski, L.: A graph library for Isabelle (2013). URL `http://www21.in.tum.de/~noschinl/documents/noschinski2013graphs.pdf`. Submitted
25. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
26. Sullivan, G.F., Masson, G.M.: Using certification trails to achieve software fault tolerance. In: FTCS, pp. 423–431 (1990)
27. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the gap: A verification framework for low-level C. In: TPHOLs, *LNCS*, vol. 5674, pp. 500–515 (2009)
28. Wirth, N.: Program development by stepwise refinement. CACM **14**(4), 221–227 (1971)