A Simpl Shortest Path Checker Verification

Christine Rizkallah

Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. Verification of complex algorithms with current verification tools in reasonable time is challenging. Certifying algorithms compute not only an output but also a witness certifying that the output is correct. A checker for a certifying algorithm is a simple program that decides whether the witness is correct for a particular input and output. Verification of checkers is feasible and leads to trustworthy computations. In previous work, we verified checkers from the algorithmic library LEDA using the interactive theorem prover Isabelle/HOL as a backend to the automatic code verifier VCC. More recently, we showed that verification can be carried out completely within Isabelle/HOL and compared this to the previous approach. We concluded that the more recent approach is more trustworthy with comparable efforts. Here we implement a shortest path checker algorithm for graphs with nonnegative edge weights in the imperative pseudocode language Simpl. Moreover, we use the more recent framework and formally verify the checker using Isabelle/HOL.

1 Introduction

A user of an algorithm has in general no means to know whether a result computed by this algorithm is correct or has been compromised by a bug. A *certifying algorithm* [3, 21, 11] produces with each output a *certificate* or *witness* that the *particular output* is correct. The accompanying *checker* inspects the witness and accepts it if the witness proves that y is a correct output for input x. Otherwise, the checker rejects the output or witness as buggy. Certifying algorithms are a key design principle of the algorithmic library LEDA [12]: Checkers are an integral part of the library and are optionally invoked after every execution of a LEDA algorithm. Adoption of the principle greatly improved the reliability of the library. Formal verification of checkers is feasible [1, 16] and further improves the reliability of certifying algorithms. Verification of checkers implies instance correctness for accepted computations of the corresponding certifying algorithm. Verifying instance correctness is orthogonal to verifying that a particular algorithm is correct.

Formal verification of a certifying computation requires establishing two theorems along the following lines: First, if the witness has a certain property, then it proves that y is a correct output for x. Second, the checker program accepts the witness if and only if it has the desired property. We provide more details in Section 2. In recent work [1] we provided a framework for verifying certifying computations. It does so by combining the concept of certifying algorithms with methods for code verification and theorem proving. More precisely, it uses the interactive theorem prover Isabelle/HOL as a backend to the automatic code verifier VCC. The first type of theorem is proved in

Isabelle/HOL and the second type of theorem is proved in VCC. The two theorems are linked by transferring statements from VCC to Isabelle/HOL. The framework is illustrated on several examples in the domain of graphs from the algorithmic library LEDA. Namely, on the connected components checker, the shortest path (with nonnegative edge weights) checker, and the maximum cardinality matching checker.

More recently we suggested establishing both theorems and hence the complete verification of checker algorithms and their implementations within Isabelle/HOL [16]. We demonstrated the approach on the connected components checker and the non-planarity checker.

Here we re-verify the shortest path checker algorithm with nonnegative edge weights solely within Isabelle/HOL. We implement the checker in Simpl, a generic imperative programming language whose semantics is formulated in Isabelle and for which a verification environment exists in Isabelle/HOL [19]. Then we verify the checker correctness within Isabelle/HOL. The complete implementation and verification is online: http://www.mpi-inf.mpg.de/∼crizkall/Publications/VerificationSimplShortestPath.zip. This folder also contains the Simpl formalization [20], the Isabelle graph library [15] and a formal proof of the witness property of the shortest path checker [18] which are all used in the verification and are also available from Isabelle's archive of formal proofs.

The verification of the shortest path (with general edge weights) checker algorithm is currently still work in progress. We implemented the more general checker in Simpl and we are about half way through with the verification.

2 Preliminaries

As described in [1], we consider algorithms taking an input from a set X and producing an output in a set Y and a witness in a set W. The input $x \in X$ is supposed to satisfy a precondition $\varphi(x)$ and the input together with the output $y \in Y$ is supposed to satisfy a postcondition $\psi(x, y)$. A *witness predicate* for a specification with precondition φ and postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$ with the following *witness property*:

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

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 fulfils the witness property. More precisely, let $x = i_X(\overline{x})$, $y = i_Y(\overline{y})$, and $w = i_W(\overline{w})$. If $\neg\varphi(x)$, C may do anything (run forever or halt with an arbitrary output). If $\varphi(x)$, C must halt and either accept or reject. It is supposed to accept if $\mathcal{W}(x, y, w)$ holds and supposed to reject otherwise. The following proof obligations arise:

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

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

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

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

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

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

2.1 Tools

Isabelle/HOL [13] 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 [6], guarantees correctness as long as the inference kernel is correct. Isabelle/HOL comes with a rich set of already formalized theories, among which are natural numbers and integers as well as sets, finite sets and as a recent addition graphs [15].

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

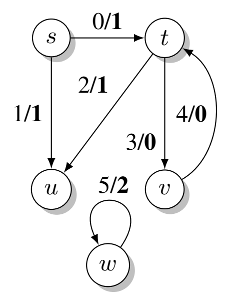
Simpl [19] is a generic imperative language designed to allow a deep embedding of real programming languages, for example C, in Isabelle/HOL for the purpose of program verification. Simpl provides the usual language constructs like functions, variable assignments, sequential compositions, conditional statements, while loops, and exceptions. Simpl does not provide the common return statement for abrupt termination, it can however be emulated by exceptions. Programs may be annotated by invariants. Simpl does not have its own expression language, expressions in Simpl programs are arbitrary Isabelle expressions. It is therefore convenient for writing pseudo-code. Specifications for Simpl programs are given as Hoare triples, where pre- and post-condition are arbitrary Isabelle expressions. In such specifications, we need to distinguish between Isabelle's logical variables and program variables (i.e., variables referring to the program state). A logical variable named x is written as x, a program variable referring to the current state as x.

3 Shortest Path Checker

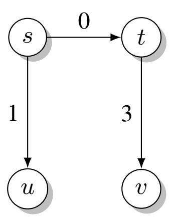
The single-source shortest-paths problem (with nonnegative edge weights) for directed graphs can be solved for instance by Dijkstra's algorithm [12, Sections 6.6 and 7.5].

Instead of verifying this algorithm, we request that it returns, along with the computed shortest distances from s to every vertex of a graph, the corresponding shortest path tree as witness. That is, we instantiate our general framework as follows:

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



(a) A directed graph G



(b) A shortest-path tree of G

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

(c) Tree representation

Fig. 1: A directed graph $G = (V, E)$ with the edges labeled $i/\mathbf{k}$, where i is a unique edge index and $\mathbf{k}$ is the cost of that edge is presented in (a). In (b) we give a shortest-path tree of G that is rooted at start vertex $s \in V$. The tree is encoded by $parent\text{-}edge$, num, and $dist$ according to the table in (c). Observe that vertex w is not reachable from s and that the cycle $t \to v \to t$ has cost zero.

Figure 1 shows a directed graph G and a shortest-path tree of G rooted at s. We encode a shortest-path tree by the functions $parent\text{-}edge$, $dist$, and num. For each v reachable from s, $dist(v)$ is the shortest-path distance from s to v and $num(v)$ is the depth of v in the shortest-path tree. For vertices v that are not reachable from s, $dist(v) = num(v) = \infty$. For reachable vertices v different from s the edge $parent\text{-}edge(v)$ is the last edge on a shortest path from s to v. The precondition $\varphi(x)$ and witness predicate $\mathcal{W}(x, y, w)$ could be summarized by the following properties:

is_wellformed: The graph G is wellformed with finite sets of vertices and edges.
non_neg_cost: For all edges e in G, $c(e) \geq 0$ where c is the cost function.
s_in_G: The source vertex s is in G.
start_val: For the source vertex s, $dist(s) = 0$.
no_path: For a vertex v in G, $dist(v) = \infty$ if and only if there is no path to v.

trian: For all edges (u, v) in G, $dist(u) + c(u, v) \geq dist(v)$.
just: If (u, v) is the parent edge of v, $dist(u) + c(u, v) = dist(v)$.

We start by presenting the witness property of the checker. We have an Isabelle locale which is equivalent to $(\varphi(x) \wedge \mathcal{W}(x, y, w))$ and a theorem stating that under the locale assumptions $\psi(x, y)$ holds. Then we present an excerpt of the Simpl checker implementation. The checker is supposed to accept if and only if the locale assumptions hold. Finally, we present part of the correctness proof of the checker. We prove that the checker terminates and that the result of the checker is equivalent to the Isabelle locales, and thus the checker accepts if and only if $\psi(x, y)$ holds. The complete implementation and proofs can be found online:
http://www.mpi-inf.mpg.de/∼crizkall/Publications/VerificationSimplShortestPath.zip.

Witness Property This formalization builds on the Isabelle graph library [15] and can be found in Isabelle's archive of formal proofs [18]. The *shortest-path-non-neg-cost* locale contains exactly the properties summarizing the precondition and the witness property that we stated earlier in the section. The theorem *correct-shortest-path* (see Listing 1) states that under the *shortest-path-non-neg-cost* locale assumptions, for any vertex v in G, $dist(v)$ is equal to the correct shortest path distance $\mu\ c\ s\ v$ from s to v using the cost function c.

Implementation We begin by fixing the types we use for the Simpl implementation (see Listing 1) and represent a graph the same way as in [1]. The type *IGraph* represents a graph G by the number of vertices *ivertex-cnt* G, number of edges *iedge-cnt* G, and a function *iedge-cnt* G mapping from edge ids to edges. Vertices of G range over the set $\{0, \ldots, (ivertex\text{-}cnt\ G) - 1\}$. Edges ids range over the set $\{0, \ldots, (iedge\text{-}cnt\ G) - 1\}$, and edges are pairs of vertices and are obtained using the function *iedge-cnt* G. If the two vertices of each edge belong to the graph we call the graph *wellformed*.

For each of the properties in the locales, we have a procedure checking this property and returning *True* if and only if it holds. For example the annotated procedure *is-wellformed* in Listing 1 checks wether a graph is wellformed. The procedure loops over edge ids in the graph and checks whether the endpoints of the corresponding edges are within the range of vertices in the graph. We add a loop invariant *is-wellformed-inv* to help with the verification. It states that the result *R* of the procedure is *True* if and only if up to step *i* in the loop all edges with edge ids less than *i* have their endpoints in the graph. The keyword ***VAR MEASURE*** in the implementation (see Listing 1) guides the automated tools in Isabelle to prove termination automatically.

Checker Correctness We prove the checker implementation terminates. The termination arguments are all trivial (loops counting upwards to some constant). The function *abs-Graph* takes a concrete graph and converts it to an abstract graph. The lemma *is-wellformed-spec* (see Listing 1) states that the procedure *is-wellformed* accepts if and only if the invariant *is-wellformed-inv(G, (iedge-cnt G))* evaluates to true. We then show that the invariant holds if and only if the abstract graph *abs-Graph(G)* is

```
theorem (in shortest-path-non-neg-cost) correct-shortest-path:
 assumes v ∈ verts G shows dist v = μ c s v

type_synonym IVertex = nat
type_synonym Edge-Id = nat
type_synonym IEdge = IVertex × IVertex
type_synonym IGraph = nat × nat × (Edge-Id ⇒ IEdge)

definition is-wellformed-inv :: IGraph ⇒ nat ⇒ bool where
 is-wellformed-inv G i ≡
 ∀ k < i. ivertex-cnt G > fst (iedges G k) ∧ ivertex-cnt G > snd (iedges G k)

procedures is-wellformed (G :: IGraph | R :: bool)
 where i :: nat, e :: IEdge
 in ANNO G. {| G = G |}
    R := True ;
    i := 0 ;
    TRY
      WHILE i < iedge-cnt G
      INV {| R = is-wellformed-inv G i ∧ i ≤ iedge-cnt G ∧ G = G |}
      VAR MEASURE (iedge-cnt G − i)
      DO
       e := iedges G i ;
       IF ivertex-cnt G ≤ fst e ∨ ivertex-cnt G ≤ snd e THEN
         R := False ;
         THROW
       FI ;
       i := i + 1
      OD
    CATCH SKIP END
    {| G = G ∧ R = is-wellformed-inv G (iedge-cnt G) |}

lemma (in is-wellformed-inv-step) is-wellformed-spec:
 ∀G. Γ ⊢_t {|G = G|} R := PROC is-wellformed(G) {|R = is-wellformed-inv G (iedge-cnt G
   )|}
```

Listing 1: Excerpts from witness property, implementation, and verification of shortest paths in Isabelle/HOL. The ANNO command binds the logical variable G so it can be used in the invariant.

wellformed (which is one of the assumptions in the $shortest\text{-}path\text{-}non\text{-}neg\text{-}cost$ locale). For all other procedures we show that their results are equivalent to some locale assumption (applied to the abstracted graph). Eventually we show that the checker procedure is equivalent to the locale. By this we conclude our proof.

4 Related Work, Conclusion and Future Work

Verifying imperative code within interactive theorem provers is a an active field of research. A semantics of C was formalized in HOL [14], and a semantics of a subset of C, called C0, was formalized in Isabelle/HOL [9]. A verification environment for the imperative language Simpl was developed within Isabelle and C0 was embedded into it [19]. This work has been applied to verify a compiler for C0 [17] and extended to verify the seL4 microkernel which is written in low-level C [8]. Simpl was also used in the Verisoft project [22]. Coq [2] was used both for programming the CompCert compiler and for proving its correctness [10]. CFML is a verification tool, embedded in Coq, that targets imperative Caml programs [5]. It has been applied to verify imperative data structures such as mutable lists, sparse arrays and union-find.

Shortest-path algorithms, especially imperative implementations thereof, are popular as case studies for demonstrating code verification [5, 4]. They target full functional correctness as opposed to instance correctness. Verifying checkers and hence instance correctness is orthogonal to verifying that a particular shortest path algorithm is correct. It can be combined with any shortest-path algorithm that is instrumented to provide the necessary witness expected by our checker. We recently proposed a framework for verification of certifying computations through formally verifying checkers using Isabelle/HOL(and its graph library [15]) as a backend to VCC [1]. We applied this framework to verify several checkers from the domain of graph theory including a shortest path checker. We later proposed a new framework for verifying checkers solely within Isabelle/HOL [16]. This provides higher trust guarantees and could be done with comparable (if not less) effort. In this paper, we re-verify the shortest path checker algorithm with nonnegative edge weights using the new framework. We implement a checker in Simpl and verify the checker correctness within Isabelle/HOL. The implementation along with the verification is about 450 lines.

For future work, we plan to use the verification of the shortest path checker algorithm presented here and AutoCorres [7] to verify the C implementation. Moreover, we are currently verifying a more involved checker algorithm for the shortest path problem with general edge weights. We implemented the checker algorithm in Simpl and are about half way through with the verification.

Acknowledgement Special thanks to Kurt Mehlhorn and Fatemeh Shirazi for their feedback on the paper.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. J. Autom. Reasoning **52**(3), 241–273 (2014)

2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
3. Blum, M., Kannan, S.: Designing programs that check their work. In: Symposium on Theory of Computing, pp. 86–97. ACM (1989)
4. Böhme, S., Leino, K.R.M., Wolff, B.: HOL-Boogie—An interactive prover for the Boogie program-verifier. In: Theorem Proving in Higher Order Logics, *Lecture Notes in Computer Science*, vol. 5170, pp. 150–166. Springer (2008)
5. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: International Conference on Functional Programming, pp. 418–430. ACM (2011)
6. Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, *Lecture Notes in Computer Science*, vol. 78. Springer (1979)
7. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 7406, pp. 99–115. Springer (2012)
8. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. Communications of the ACM **53**(6), 107–115 (2010)
9. Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In: Software Engineering and Formal Methods, pp. 2–12. IEEE Computer Society (2005)
10. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7), 107–115 (2009)
11. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011)
12. Mehlhorn, K., Näher, S.: The LEDA Platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
13. 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)
14. Norrish, M.: C formalised in HOL. Ph.D. thesis, Computer Laboratory, University of Cambridge (1998)
15. Noschinski, L.: Graph theory. Archive of Formal Proofs (2013). http://afp.sourceforge.net/entries/Graph_Theory.shtml, Formal proof development
16. Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through AutoCorres and Simpl. In: NASA Formal Methods, pp. 46–61 (2014)
17. Petrova, E.: Verification of the C0 compiler implementation on the source code level. Ph.D. thesis, Saarland University, Saarbrücken (2007)
18. Rizkallah, C.: An axiomatic characterization of the single-source shortest path problem. Archive of Formal Proofs (2013). http://afp.sourceforge.net/entries/ShortestPath.shtml, Formal proof development
19. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
20. Schirmer, N.: A sequential imperative programming language syntax, semantics, Hoare logics and verification environment. Archive of Formal Proofs (2008). http://afp.sourceforge.net/entries/Simpl.shtml, Formal proof development
21. Sullivan, G.F., Masson, G.M.: Using certification trails to achieve software fault tolerance. In: Fault-Tolerant Computing, pp. 423–431. IEEE Computer Society (1990)
22. Verisoft. http://www.verisoft.de/ (2007)