Verification of Program Computations

Dissertation

Thesis for obtaining the title of Doctor of Engineering
of the Faculties of Natural Sciences and Technology
of Saarland University vorgelegt von

Christine Rizkallah

Saarbrücken
2015

Dean: Prof. Dr. Markus Bläser

Colloquium: 18 September 2015

Examination Board:

Supervisor: Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn

Reviewers: Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn
Prof. Dr. Tobias Nipkow

Chair: Prof. Dr. Raimund Seidel

Research Assistant: Dr. Vinay Settty

Abstract

Formal verification of complex algorithms is challenging. Verifying their implementations in reasonable time is infeasible using current verification tools and usually involves intricate mathematical theorems. Certifying algorithms compute in addition to each output a witness certifying that the output is correct. A checker for such a witness is usually much simpler than the original algorithm – yet it is all the user has to trust. The verification of checkers is feasible with current tools and leads to computations that can be completely trusted. We describe a framework to seamlessly verify certifying computations. We demonstrate the effectiveness of our approach by presenting the verification of typical examples of the industrial-level and widespread algorithmic library LEDA. We present and compare two alternative methods for verifying the C implementation of the checkers.

Moreover, we present work that was done during an internship at NICTA, Australia's Information and Communications Technology Research Centre of Excellence. This work contributes to building a feasible framework for verifying efficient file systems code. As opposed to the algorithmic problems we address in this thesis, file systems code is mostly straightforward and hence a good candidate for automation.

Zusammenfassung

Die formale Verifikation der Implementierung komplexer Algorithmen ist schwierig. Sie übersteigt die Möglichkeiten der heutigen Verifikationswerkzeuge und erfordert für gewöhnlich komplexe mathematische Theoreme. Zertifizierende Algorithmen berechnen zu jeder Ausgabe ein Zerfitikat, das die Korrektheit der Antwort bestätigt. Ein Checker für ein solches Zertifikat ist normalerweise ein viel einfacheres Programm und doch muss ein Nutzer nur dem Checker vertrauen. Die Verifizierung von Checkern ist mit den heutigen Werkzeugen möglich und führt zu Berechnungen, denen völlig vertraut werden kann. Wir beschreiben eine Rahmenstruktur zur Verifikation zertifizierender Berechnungen und demonstrieren die Effektivität unseres Ansatzes an Hand typischer Beispiele aus der hochqualitätiven und oft eingesetzten LEDA Algorithmenbibliothek. We präsentieren und bewerten zwei alternative Methoden zur Verifikation von Checkerimplementierungen in C.

Desweiteren beschreiben wir Ergebnisse, die während eines Praktikums am NICTA, dem Australischen Forschungszentrum für Informations- und Kommunikationstechnik, erzielt wurden. Diese Arbeit trägt zum Aufbau einer praktisch einsetzbaren Rahmenstruktur zur Verifizierung von Code für effiziente Dateisysteme bei. Im Gegensatz zu den algorithmischen Problemen, die wir in dieser Arbeiten behandeln, ist der Code für Dateisysteme weitgehend unkompliziert und daher ein guter Kandidat zur Automatisierung.

Diese Arbeit ist in englischer Sprache verfasst.

Acknowledgments

I am indebted to my supervisor Kurt Mehlhorn for teaching me not only about science but about life in general. I would like to thank Kurt for our regular meetings that ensured I am on track, for his ideas contributing to this work, and for our interesting discussions. I thank him for having an open mind and giving me the freedom to diverge in my research topic from his main interest in Algorithms. Watching Kurt work is an invaluable lesson. It is inspiring to see how one could be a calm manager in a hectic academic setup and yet lead a very successful group. I am also grateful to Kurt for his feedback on several drafts of this thesis.

In addition to Kurt, the main topic of this thesis is done in collaboration with Eyad Alkassar, Sascha Böhme, and Lars Noschinski. I thank Eyad for initiating the project and Sascha for his collaboration and for answering all my initial Isabelle questions. Furthermore, I thank Lars for creating a useful Isabelle library for graphs, which I use in this thesis, and for verifying the Kuratowski checker using our framework further demonstrating the usability of the framework. I also thank Tobias Nipkow for his interest in the topic and for his advice regarding conferences and journals. I am delighted that Tobias accepted to act as a reviewer for my thesis.

I am grateful to Gerwin Klein for inviting me for a six month internship at NICTA and to Gernot Heiser for funding my stay. Special thanks to Toby Murray for his thorough supervision during my stay. I thank the trustworthy file systems group at NICTA for the enjoyable interaction on such a big project. It was a pleasure working with all of them. I additionally thank Thomas Sewell for helping me define a sensible state relation (described in Section 3.2.2).

I thank all those who in one way or another inspired me to pursue the subject of this thesis or initiated my interest in the field, some of whom are Chad E. Brown and Gert Smolka; they are my master thesis supervisors and the ones who taught me about logic, and Haythem O. Ismail who lectured several inspiring theoretical lectures during my undergraduate studies.

I am obliged to all the staff members of the Max Planck Institute for Informatics and the NICTA staff for maintaining a cosy homely atmosphere.

I am indebted to David Greenaway, Gerwin Klein, Toby Murray, Yutaka Nagashima, Adrian Neumann, and Lars Noschinski for their comments and feedback on parts of this thesis. Special thanks to David for his many comments that improved the readability of this thesis.

Thanks to my parents, grandparents, sister, Fateme, Young-Jun, and Veronika for their care and support and for constantly boosting my morale.

Contents

1

Introduction

One of the most prominent and costly problems in software engineering is correctness of software. This thesis describes two separate contributions. Our main work is concerned with software for difficult algorithmic problems in the domain of graphs. The algorithms for such problems are complex; formal verification of the resulting programs in reasonable time is not tractable, which explains why few graph algorithms have been verified. We give a framework for obtaining *formal instance correctness*, i.e., formal proofs that outputs for particular inputs are correct. We do so by combining the concept of certifying algorithms with methods for code verification and theorem proving. The other work was done during an internship and contributes to building a feasible framework for verifying efficient file systems code.

1.1. Contributions

Formal verification of complex algorithms is challenging. Verifying their implementations requires both intricate reasoning about their high-level algorithms, and low-level reasoning about their implementations beyond the capabilities of current verification tools. *Certifying algorithms* compute in addition to each output a *witness* certifying that the output is correct. A checker for such a witness is usually much simpler than the original algorithm – yet it is all the user has to trust. The verification of checkers is feasible with current tools and leads to computations that can be completely trusted. We take the certifying-algorithms approach a step further by developing a methodology for verifying the total correctness of such checkers. This gives us a framework to seamlessly verify certifying computations. We give two approaches for verification and investigate their trade-offs. In the first approach [Alkassar et al., 2014], we make use of the mature off-the-shelf C program verifier VCC [Cohen et al., 2009] coupled with the interactive theorem prover Isabelle/HOL [Nipkow et al., 2002]. We use VCC for establishing the correctness of the checker and Isabelle/HOL for proving high-level mathematical properties of the algorithm. We demonstrate the effectiveness of our approach by presenting the verification of typical examples of the industrial-level and widespread algorithmic library LEDA [Mehlhorn and Näher, 1999]. We consider examples from the field of graph theory, namely, the checker for connectedness of graphs, a shortest path checker, and a checker for maximum cardinality matchings in graphs. This approach has the advantage that it could be carried out with reasonable effort in 2011. In the second approach [Noschinski et al., 2014], we replace VCC with verified Isabelle/HOL tools for C code that emerged meanwhile. While these tools are less mature, they provide other advantages, notably higher soundness guarantees. Relying on a single tool provides higher soundness guarantees because we have less trusted tools in our verification chain and we no longer need to trust the translation from one tool to the other. 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 checkers for connectedness of graphs written in both C and Simpl. Moreover, we re-verify the shortest path checker written in Simpl [Rizkallah, 2014]. For checkers written in C, we translate from C to Isabelle using the AutoCorres tool set and then reason in Isabelle. For checkers written in Simpl, Isabelle is the only tool needed. This approach was also successfully used to verify the LEDA checker for non-planarity of graphs [Noschinski et al., 2014]. We conclude that the new approach provides higher trust guarantees and it is particularly promising for checkers that require domain-specific reasoning.

We also present work done during a six month internship at the Trustworthy Systems group at NICTA. This work contributes to building a feasible framework for verifying efficient file systems code. As opposed to the algorithmic problems we mainly address in this thesis, file systems code is mostly straightforward yet long and tedious. It deals with a lot of error handling cases. Filesystems are, therefore, a good candidate for code generation and proof automation.

In the remainder of this chapter, we introduce the concept of certifying algorithms and provide an overview of the tools used in this thesis, namely Isabelle/HOL, VCC, Simpl and AutoCorres.

1.2. Certifying Algorithms

A *certifying algorithm* [Blum and Kannan, 1989, Sullivan and Masson, 1990, Arkoudas and Rinard, 2005, McConnell et al., 2011, Alkassar et al., 2011b] produces with each output a *certificate* or *witness* that the *particular output* is

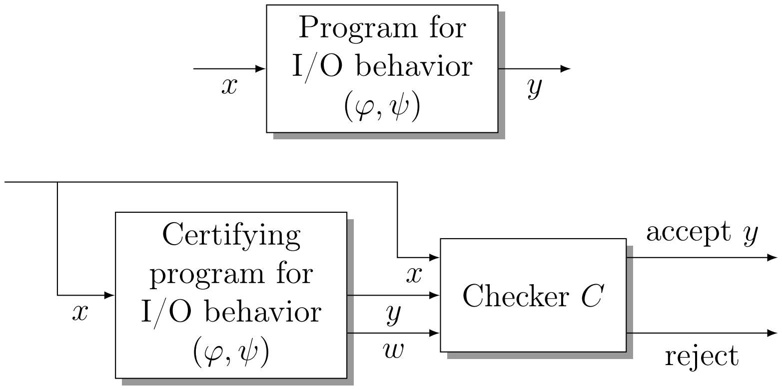


Figure 1.1. The top figure shows the input-output behavior (φ, ψ) of a conventional program. The user feeds an input x satisfying $\varphi(x)$ to the program, and the program returns an output y satisfying $\psi(x, y)$. A certifying algorithm for input-output behavior (φ, ψ) computes y and a witness w. The checker C accepts the triple (x, y, w) if and only if w is a valid witness for the postcondition $\psi(x, y)$, i.e., it proves $\psi(x, y)$.

correct. The accompanying *checker* for a certifying algorithm with input x, output y, and witness w takes as input the triple (x, y, w) and accepts the triple if w proves that y is a correct output for input x. Otherwise, the checker rejects the output or witness as buggy[1].

Figure 1.1 contrasts a standard algorithm with a certifying algorithm for input-output behavior (φ, ψ). An algorithm for input-output behavior (φ, ψ) receives an input x satisfying a precondition $\varphi(x)$ and is supposed to deliver an output y satisfying the postcondition $\psi(x, y)$. We call such a y a *correct output*. If the input does not satisfy the precondition, the result of the computation is unspecified. A user of a standard algorithm has, in general, no means of knowing that y is a correct output and has not been compromised by a bug. In contrast, if the accompanying checker of a certifying algorithm accepts, the user may proceed with the complete confidence that output y has not been compromised by a bug. If the checker rejects, either y is incorrect or w is not a proof of the correctness of y.

We illustrate the concept of certifying algorithms with an example. The greatest common divisor of two nonnegative integers a and b, not both zero, is the largest integer g that divides a and b. We write $g = gcd(a, b)$. The extended Euclidean algorithm is a certifying algorithm for greatest common divisor. In addition to the output $g = gcd(a, b)$, it also computes integers s and t such that $g = s \cdot a + t \cdot b$ as a witness.[2] The checker checks that g divides a and b and that $g = s \cdot a + t \cdot b$. Why does this prove that g is the greatest

---

[1] Throughout the thesis, we say the checker *accepts* if the checker returns *True*; otherwise, we say it *rejects*.

[2] It can be easily shown that such integers s and t always exist.

common divisor of a and b? Consider any integer d that divides a and b. Then $g = s \cdot a + t \cdot b = (s \cdot (a/d) + t \cdot (b/d)) \cdot d$, and hence, d divides g.

Another example is deciding whether a graph is bipartite. A graph is bipartite if the vertices can be colored by colors red and blue such that the endpoints of every edge have distinct colors. The output of a non-certifying algorithm is either YES or NO. A certifying algorithm may output a two-coloring in the YES-case and an odd-length cycle contained in the graph in the NO-case. An odd-length cycle can clearly not be two-colored and hence any graph containing an odd-length cycle cannot be two-colored. The checker proceeds as follows. In the YES-case, it iterates over all edges of the graph and checks that the endpoints have distinct colors. In the NO-case, it checks that all edges of the cycle are present in the graph and that the cycle has odd length.

Certifying algorithms are a key design principle of the algorithmic library LEDA [Mehlhorn and Näher, 1999]: Checkers are an integral part of the library and may (optionally) be invoked after every execution of a LEDA algorithm. The adoption of this principle greatly improved the reliability of the library. However, how can one be sure that the checker programs are correct? Kurt Mehlhorn used to answer: “Checkers are simple programs with little algorithmic complexity. Hence, one may assume that their implementations are correct.” We give a better answer in this thesis.

We take the certifying-algorithms approach a step further by developing a methodology to verify the total correctness of the checkers. We demonstrate it on several checkers from the domain of graphs. We compare two alternative methods for verifying the C implementation of the checkers.

1.3. Tools

We introduce the main tool used in this thesis, the interactive theorem prover Isabelle/HOL. It is used for proving all the high level mathematical properties of the checkers. We then describe other tools, VCC, an automatic code verifier which is used for verifying the C implementation of the checkers; Simpl, a generic imperative language embedded in Isabelle which is used to implement the checkers (in addition to C); and AutoCorres, which is used to simplify the verification of the C checkers within Isabelle/HOL.

1.3.1. Isabelle/HOL

Isabelle/HOL [Nipkow et al., 2002] is an interactive theorem prover for classical higher-order logic based on Church’s simply-typed lambda calculus.[3] 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

---

[3] In this work, we use Isabelle 2014.

proof methods) ultimately rely on these rules only. This approach (called the LCF approach, due to Edinburgh LCF, which pioneered the idea [Gordon et al., 1979]) 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 directed graphs [Noschinski, 2014]. The graph library that we use in our formalization supports general infinite directed graphs with potential labeled and parallel arcs.

Isabelle/HOL supports new types, which can be introduced by defining them as records (isomorphic to tuples with named update and selector functions), among other means. New constants can be introduced, for example, via definitions relative to already existing constants. There is a distinction between the meta logic and the object logic. In the meta logic, the symbol $\bigwedge$ stand for universal quantification and $\Longrightarrow$ stands for implication. The notation $[\![P1; \ldots; P]\!] \Longrightarrow Q$ is short hand for for $P1 \Longrightarrow \ldots \Longrightarrow Pn \Longrightarrow Q$. Through out the thesis we use the Isabelle notation $xs@ys$ for the concatenation of two lists xs and ys, and $x\#xs$ stands for a list with head x and tail xs.

Proofs in Isabelle/HOL can be written in a style named Isabelle/Isar, which is close to that of mathematical textbooks. In this style, 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 are made.

In Isabelle, theorems can be proven in the context of a *locale*. A *locale declaration* consists of constant declarations and assumptions. Theorems proven in the *context* of a locale can use the constants and implicitly 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. The notation + in a locale stands for locale inheritance.

1.3.2. VCC

VCC [Cohen et al., 2009] is an assertional, automatic, deductive code verifier for full C code. Specifications in the form of function contracts, data invariants, and loop invariants as well as further annotations to maintain inductively defined information or to guide VCC otherwise, are added directly into the C source code as comments. During builds with a C compiler, these annotations are ignored. From the annotated program, VCC generates verification conditions for partial or total correctness, which it then tries to discharge using the automatic theorem prover Z3 [de Moura and Bjørner, 2008] or through the Boogie verifier [Barnett et al., 2006].

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

1.3.3. Simpl

A program can be written in a theorem prover either as a *deep embedding* in terms of syntax (e.g, defined by using a datatype) or it can be written as a *shallow embedding* in terms of functions in the logic of the theorem prover (e.g, higher-order logic) [Myreen, 2012]. Shallow embeddings are easier to reason about, however using deep embeddings allow reasoning about the program structure inductively [Myreen, 2012]. Using a deep embedding is inevitable if one wants to verify programs written in a particular syntax e.g, an imperative language or in the C programming language. See [Wildmoser and Nipkow, 2004] for a further discussion about deep embeddings versus shallow embeddings in Isabelle.

Simpl [Schirmer, 2006] 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 [Norrish, 2012] 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, i.e., expressions are shallowly embedded in Simpl. 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.

1.3.4. Autocorres

The C-to-Isabelle parser makes no effort to abstract from details of the C-language. AutoCorres [Greenaway et al., 2012] 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:

$('\mathrm{s},\alpha)$ *nondet-monad* $= '\mathrm{s} \Rightarrow (\alpha \times '\mathrm{s})$ *set* $\times$ *bool*

In this monad, program statements are a function from a heap to a tuple consisting of a failure flag and the nondeterministic 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 [Cock et al., 2008].

2

Verification of Certifying Computations

This chapter is based on the following publications [Alkassar et al., 2011a, Alkassar et al., 2014, Noschinski et al., 2014, Rizkallah, 2014]. In Section 2.1, I describe the verification framework developed in collaboration with Eyad Alkassar, Sascha Böhme, and Kurt Mehlhorn. In Sections 2.2, 2.3.2, and 2.3.3, I explain my contribution that consists of formalizations and proofs in Isabelle/HOL. In Section 2.3.1, I summarize work done by Eyad Alkassar and Sascha Böhme on verifying the C implementation of checkers using VCC and on exporting proof obligations from VCC to Isabelle/HOL. Appendix A presents the Isabelle/HOL theories that are relevant to this chapter. The theory files are also available online [Rizkallah, 2015].

2.1. Outline of Methodology

We consider algorithms that take an input from a set X and produce an output in a set Y as well as a witness in a set W. The input $x \in X$ satisfies a precondition $\varphi(x)$, and the input together with the output $y \in Y$ satisfies, assuming that the algorithm is bug free, a postcondition $\psi(x, y)$. A *witness predicate* for a specification with precondition φ and postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$, where W is a set of witnesses with the following *witness property*:

$$\varphi(x) \wedge \mathcal{W}(x, y, w) \longrightarrow \psi(x, y). \tag{2.1}$$

In contrast to algorithms that work on abstract sets X, Y, and W, the implementing programs 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 the mappings $i_X : \overline{X} \rightarrow X$, $i_Y : \overline{Y} \rightarrow Y$, and $i_W : \overline{W} \rightarrow W$.

We illustrate these definitions through the example from the previous chapter. In the case of greatest common divisors, $X = W = \mathbb{Z} \times \mathbb{Z}$ and $Y = \mathbb{Z}$. For input (a, b), output g and witness (s, t), the precondition $\varphi((a, b))$ states that the inputs a and b are nonnegative integers and that at least one of them is not zero. The postcondition $\psi((a, b), g)$ states that $g = gcd(a, b)$. The witness predicate $\mathcal{W}((a, b), g, (s, t))$ states that $g = sa + tb$ and that g divides a and b. A typical representation of integers in implementations is via bitstrings. Hence, $\overline{X}$ and $\overline{W}$ are each the set of pairs of bit strings, and $\overline{Y}$ is the set of bitstrings. The mappings i_X, i_Y, and i_W map (pairs of) bit strings to the corresponding (pairs of) integers.

The checker program C receives a triple $(\overline{x}, \overline{y}, \overline{w})$ and assuming that C is correctly implemented, it decides whether $(\overline{x}, \overline{y}, \overline{w})$ 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 (run forever or halt with an arbitrary output). If $\varphi(x)$, C must halt and either accept or reject. The checker C is required to accept if $\mathcal{W}(x, y, w)$ holds and is required to reject otherwise. In order to achieve formal instance correctness, our approach is to prove the following two proof obligations: *Checker Correctness* and *Witness Property* which together imply our final correctness property.

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

1. If $\varphi(x)$, C halts.
2. If $\varphi(x)$ and $\mathcal{W}(x, y, w)$, C accepts $(\overline{x}, \overline{y}, \overline{w})$, and if $\varphi(x)$ and $\neg\mathcal{W}(x, y, w)$, C rejects the triple.

Witness Property: $\varphi(x) \wedge \mathcal{W}(x, y, w) \longrightarrow \psi(x, y)$.

In our running example, the witness property is

$$a + b > 0 \wedge g|a \wedge g|b \wedge g = a \cdot s + b \cdot t \longrightarrow g = gcd(a, b).$$

Here a, b, g, s and t are assumed to be nonnegative integers.

Once we have proven these two properties, we can then prove our final correctness theorem, as follows:

---

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

Theorem 2.1. *Assume that a checker C satisfies the checker correctness property and a witness predicate $\mathcal{W}$ satisfies the witness property. 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 $(\overline{x}, \overline{y}, \overline{w})$, $\varphi(x) \longrightarrow \psi(x, y)$. If C rejects a triple $(\overline{x}, \overline{y}, \overline{w})$, $\varphi(x) \longrightarrow \neg\mathcal{W}(x, y, w)$.

Proof. If C accepts $(\overline{x}, \overline{y}, \overline{w})$, we have $\varphi(x) \longrightarrow \mathcal{W}(x, y, w)$ by the correctness proof of C. Then by (2.1) we have a formal proof for $\varphi(x) \longrightarrow \psi(x, y)$. Conversely, if C rejects the triple, the correctness proof of C establishes $\varphi(x) \longrightarrow \neg\mathcal{W}(x, y, w)$. □

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

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

The existence of a witness is part of the correctness argument of the solution algorithm (e.g., the shortest-path algorithm, the maximum-matching algorithm). As previously mentioned, we do not verify the solution algorithms. Rather, the execution of the solution algorithm establishes the existence of a witness whenever it is called for a specific input $\overline{x}$. It returns $\overline{y}$ and $\overline{w}$, which we then hand to the checker C. In this way, we obtain formal instance correctness without having to verify the solution algorithm. Of course, this leaves the possibility that the solution algorithm is incorrect and does not always provide a $\overline{y}$ and $\overline{w}$ such that the checker accepts $(\overline{x}, \overline{y}, \overline{w})$.

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

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

We discuss next how to fulfill the two stated proof obligations, the checker correctness and the witness property, in a *comprehensive* and *efficient* framework. Comprehensive means that the final proof formally combines (as much as possible at the syntactic level) the correctness arguments for all levels (implementation, abstraction, and mathematical theory). Efficient means we are able to carry out our proofs in a reasonable amount of time. For example, applying a general theorem prover with no extra tool assistance to verify imperative code, while being comprehensive, would involve a lot of language-specific overhead and lead to less automation. Similarly, a specialized code verifier, while efficient, is often not powerful enough to cover nontrivial mathematical properties. The goals of comprehensiveness and efficiency often conflict because different tools usually come with different languages, axiomatization sets, etc.

LEDA Checkers We are interested in verifying checkers from the widely used algorithmic library LEDA. LEDA is written in C++ [Mehlhorn and Näher, 1999]. Our aim is to verify code which is as close as possible to the original implementation. By this, we demonstrate the feasibility of verifying already established libraries written in imperative languages such as C. We give two alternative approaches for verifying C checkers. Formally verifying the C++ implementations remains an open problem.

Overview We first present several case studies from LEDA in the domain of graph theory, namely, connected components, single source shortest paths with non-negative edge costs, single-source shortest paths with arbitrary edge-costs, and maximum cardinality matchings in graphs. We formally prove that the witness properties of those examples is correct in Isabelle/HOL. Then we propose two alternative approaches for verifying checker correctness; the *VCC approach* and the *AutoCorres approach*. We initially proposed the VCC approach that suggests using Isabelle as a backend to VCC. It uses second-order logic as a common interface language between VCC and Isabelle. Meanwhile, an Isabelle tool called AutoCorres emerged, that simplifies reasoning about C within Isabelle. We therefore later on proposed the AutoCorres approach, because AutoCorres made it feasible to use Isabelle/HOL for the entire verification. We demonstrate the AutoCorres approach on the connected components example. The AutoCorres approach was also successfully used by Lars Noschinski to verify a more involved checker for graph non-planarity [Noschinski et al., 2014].

The VCC approach We verify the code with VCC [Cohen et al., 2009], an automatic code verifier for full C. Our choice of VCC was motivated by the maturity of the tool and the provision of an assertion language that is rich enough for our requirements. In the Verisoft XT project [Verisoft XT, 2010], VCC was successfully used to verify tens of thousands of lines of C code. The assertion language offers ghost code and ghost types such as maps and unbounded integers. This gives enough expressiveness to quantify over graphs, labelings, etc., and simplifies the translation to other proof systems. For verifying the mathematical part, we use Isabelle/HOL because of the large amount of already formalized mathematics, its descriptive proof format, and its various automatic proof methods and tools.

Checker Verification: The starting point is the checker code written in C.

Using VCC, we annotate the functions and data structures such that the witness predicate $\mathcal{W}$ can be established as the postcondition of the checker function. We define the witness predicate and the pre- and postcondition as well as the mappings i_X, i_Y, and i_W as pure mathematical objects using VCC ghost types and ghost functions. Note that as a precondition to all our programs we assume that the concrete input values $\overline{x}$ are valid (i.e., they are not NULL pointers and do not point outside array bounds nor to memory protected addresses nor outside the address space). This is ensured by using the VCC invariant `\wrapped` or other invariants that ensure that the objects are owned by the current thread.

Export to Isabelle/HOL: Establishing the witness property involves, in general, mathematical reasoning beyond what is conveniently done in VCC. We therefore translate the precondition, witness predicate, postcondition, and the abstract representations of the input, output, and witness from VCC to Isabelle/HOL. Since we formulated them as pure mathematical objects in VCC, this translation is purely syntactical and does not involve any VCC specifics. While in our work this translation was carried out manually, this step could easily be automated.

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

We stress that using this approach the overall correctness theorem, i.e., the witness property, can be formulated in VCC; this is important for usability. The user of a verified checker only has to look at its VCC specification; the fact that we outsource the proof of the witness property to Isabelle/HOL is of no concern to the user. Once proven in Isabelle/HOL, we may then formulate the witness property as an axiom in VCC. This is sound since we restrict the language for describing the witness property to second-order logic, which guarantees that we can express it equivalently in Isabelle's higher-order logic (see Section 2.3.1). More precisely, since the VCC formulation of the witness property is valid if and only if its translation to Isabelle is valid, and since Isabelle is consistent, and hence, only valid statements can be proven, it is sound to add the witness property as an axiom to VCC.

The AutoCorres Approach AutoCorres makes it feasible to reason about the C code implementation of the checker directly within Isabelle/HOL. In this approach, both the witness property and the checker correctness proof obligations are discharged using Isabelle/HOL. Hence, the overall correctness theorem is established in Isabelle/HOL. This approach requires trusting a smaller code base which leads to more trustworthy results. More precisely, in addition to the

Isabelle kernel, in this approach we only trust the C-to-Isabelle parser, which is quite small. In the VCC approach, we also relied on the correctness of the translation between VCC and Isabelle, the VCC engine which consists of a large code base, and an automatic theorem prover, called Z3, that is used by VCC. Furthermore, using only one system saves us from having to duplicate formalization effort in two systems and having to export theorems from one system to another.

We demonstrate our methodology on a number of LEDA checkers from the domain of graph theory. In the next section, we describe the checkers as case studies and explain the proofs of their witness properties.

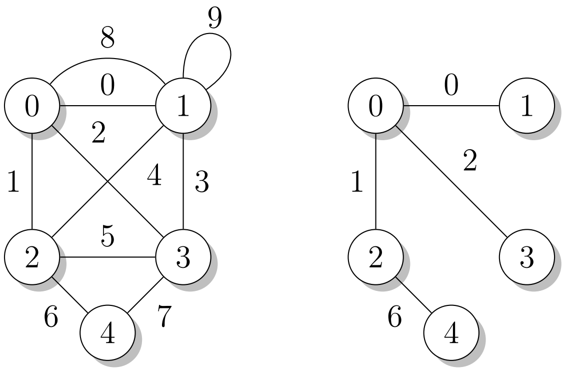
2.2. Case Studies and Witness Properties

We present a number of checkers and explain how they fit into our framework. Moreover, we present the Isabelle/HOL formalization of the witness predicates of the checkers and explain the proofs of their witness properties.

2.2.1. Connected Components

Our first case study considers the connected components problem. 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 [Mehlhorn and Näher, 1999, Section 7.4]. In the negative case, i.e., when the graph is not connected, there is a simple witness. It consists of a cut S, 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. In other words, no edge crosses the cut. In the positive case, i.e., when the given graph is connected, the algorithm can produce a spanning tree of G as a witness. A spanning tree of G is a subgraph of G, which is a tree and contains all vertices of G. On a high level, we instantiate our general approach 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
$\varphi(x)$ = V and E are finite sets and G is wellformed i.e., a pair of vertices in $V \times V$ is associated with every $e \in E$
$\mathcal{W}(x, y, w)$ = y is *True* and w is a spanning tree of G, 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.



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

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

(c) Spanning tree representation

Figure 2.1. An example of a connected graph G and a spanning tree of G witnessing its connectivity. The vertices belong to the set $\{0, \ldots, n-1\}$ and the edges are pairs of vertices indexed by an identifier ranging from 0 to $m-1$, where n and m are the number of vertices and edges in G. The spanning tree in (b) can be represented by a root vertex $r = 0$ and functions *parent-edge* and *num* as shown in the table in (c). Graphs may have self-loops and parallel edges.

We restrict ourselves to the positive case $y = True$. We describe a checker for the spanning tree witness and the verification of this checker. Figure 2.1 shows a graph G and its spanning tree. We represent spanning trees by functions *parent-edge* and *num* and by a root vertex r, and we view the edges of the tree oriented towards r: for $v \neq r$, *parent-edge*(v) is the first edge on the path from v to r, *parent-edge*$(r) = \perp$, and *num*(v) is the length of the path from v to r for all v. The function *num* is needed in order to show that *parent-edge* encodes a forest.

Proving the Witness Property

We prove in Isabelle that a spanning tree witnesses the connectivity of a graph. The proof is done in two steps. The first step is a high-level proof in which we abstract from concrete representations of graphs and spanning trees.

Our formalization builds on the Isabelle graph library developed by Lars Noschinski [Noschinski, 2014]. Graphs in this library are directed. A *fin-digraph* is a wellformed directed graph with a finite set of vertices and a finite set of edges; the library reserves the word *digraph* for graphs without parallel edges and self-loops. In Isabelle we represent undirected graphs as bidirected graphs[2], i.e., directed graphs containing for every edge (u, v) also the reversed edge (v, u). The function *mk-symmetric* maps a fin-digraph to a bidirected fin-digraph by appropriately extending the set of edges with missing reversed edges. A vertex v

---

[2] We do so in order to directly use the Isabelle graph library.

```
locale connected-components-locale = fin-digraph +
  fixes num :: α ⇒ nat
  fixes parent-edge :: α ⇒ β option
  fixes r :: α
  assumes r-assms: r ∈ verts G ∧ parent-edge r = None ∧ num r = 0
  assumes parent-num-assms:
    ⋀v. v ∈ verts G ∧ v ≠ r ⟹
      ∃e ∈ arcs G.
        parent-edge v = Some e ∧
        head G e = v ∧
        num v = num (tail G e) + 1
```

Listing 2.1. The *fin-digraph* locale assumes the directed graph G is well-formed and finite. The locale *connected-components-locale* inherits from *fin-digraph*, meaning it includes all constants and assumptions in *fin-digraph*, and additionally includes the assumptions *r-assms* and *parent-num-assms*.

is *reachable* from a vertex u in a (bi)directed graph G if there exists a directed *walk* from u to v in G, i.e., a sequence $(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)$ of edges with $u_1 = u$, $v_k = v$, and $v_i = u_{i+1}$ for $1 \leq i < k$. An alternative and equivalent formalization of reachability between vertices u and v in G is via sequences of vertices $v_1, v_2, \ldots, v_k$, where $v_1 = u$, $v_k = v$ and (v_i, v_{i+1}) is an edge of G for $1 \leq i < k$. We say a vertex v is *reachable through a path* from a vertex u in G if v is reachable from u through a path in G. An undirected graph is *connected* if for any two vertices of the graph, one is reachable through a path from the other.

Our high-level proof rests on the Isabelle locale *connected-components-locale* (Listing 2.1) that describes the assumptions of our theorem. We fix G to be a fin-digraph where α is an abstraction of the type of vertices and β is an abstraction of the type of edges. Furthermore, we fix a representation of spanning trees with functions *parent-edge* and *num* and vertex r as the root. Based on these assumptions we prove that G is connected. We first show that every vertex v in the graph is reachable from the root r by induction on *num* v, i.e., the length of the walk from r to v in the spanning tree. The base case follows directly from our assumptions. For the inductive step, we can assume a walk from r to the parent of a vertex v. Using the assumptions, this walk can be extended to a walk from r to v since there is an edge between v and its parent. Now, since G is bidirected, we can establish that there is a walk between any two vertices of G by combining the walks that connect them with the root r. If there is a walk between two vertices, there is also a path between them. Therefore, all vertices in G are reachable through a path from one another, and hence, G is connected.

2.2.2. Shortest Path

In graph theory, a *shortest path* is a path between two vertices in a graph such that the sum of the costs of its constituent edges is minimized. The single-source shortest path problem is the problem of finding shortest-paths from a source vertex in the graph to all vertices in the graph. The single-source shortest-paths problem (with nonnegative edge costs) for directed graphs can be solved for instance by Dijkstra's algorithm [Mehlhorn and Näher, 1999, Sections 6.6 and 7.5]. Instead of verifying the algorithm directly, we request that it returns, not only the computed shortest distances from s to every vertex of the graph, but also the corresponding shortest path tree as witness.

We instantiate our general framework as follows:

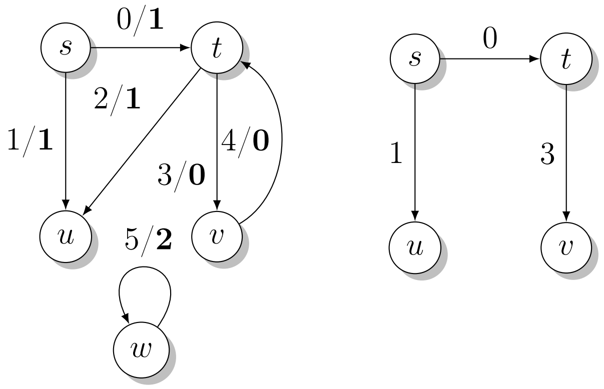
input x = a directed graph $G = (V, E)$, a function $c : E \to \mathbb{N}$ for edge costs, a vertex s
output y = a mapping $dist : V \to (\mathbb{N} \cup \infty)$
witness w = a tree rooted at s
$\varphi(x)$ = $s \in V$
$\mathcal{W}(x, y, w)$ = G is wellformed and w is a shortest-path tree, i.e., for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

Figure 2.2 shows a directed graph and a shortest-path tree rooted at s. We encode a shortest-path tree by functions *parent-edge*, *dist*, and *enum*[3]. For each v reachable from s, $dist(v)$ is the shortest-path distance from s to v and $enum(v)$ is the depth of v in the shortest-path tree. For vertices v that are not reachable from s, $dist(v) = enum(v) = \infty$. For reachable vertices v different from s, the edge $parent\text{-}edge(v)$ is the last edge on a shortest path from s to v. This witness is somewhat verbose. As we will see in the explanation of the proof of correctness of the witness property, we could do without the *parent-edge* function. If all edge costs are positive, no witness is required beyond the *dist* function. If one also allows cost zero for edges as we do, the depth function *enum* is indispensable [McConnell et al., 2011, Section 2.4].

Let $\mu(c, s, v)$ be the shortest path distance from the source vertex s to a vertex $v \in V$ using the cost function c, i.e., $\mu(c, s, v) = inf\{c(p);\ p$ is a walk from s to $v\}$ where inf is the infimum of a set and $c(p)$ is the cost of path p. For unreachable vertices v, $\mu(c, s, v) = \infty$. For reachable vertices v, if there exists a

---

[3] Unlike the function $num : V \to \mathbb{N}$ in the previous section, $enum : V \to \mathbb{N} \cup \{\infty\}$ and this is the reason we call it differently.



(a) A directed graph G **(b)** A shortest-path tree of G

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

(c) Tree representation

Figure 2.2. A directed graph $G = (V, E)$ with the edges labeled $i/\mathbf{k}$, where i is a unique edge index and where $\mathbf{k}$ is the cost of that edge, and a shortest-path tree of G rooted at start vertex $s \in V$. The tree is encoded by *parent-edge*, *enum* 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.

walk from s to v that includes a *negative cycle* (a cycle of negative total cost) then $\mu(c, s, v) = -\infty$[4]. For all other reachable vertices v, $\mu(c, s, v) \in \mathbb{R}$.

The precondition $\varphi(x)$ and witness predicate $\mathcal{W}(x, y, w)$ can be summarized by the conjunction of the following properties:

fin-digraph: The directed graph G is wellformed with finite sets V and E.

We partition V into three sets

$$\begin{aligned}
&V_\infty = \{v.\ v \in V \wedge \mathit{dist}(v) = \infty\},\\
&V_{-\infty} = \{v.\ v \in V \wedge \mathit{dist}(v) = -\infty\}, \text{ and}\\
&V_f = \{v.\ v \in V \wedge \mathit{dist}(v) \in \mathbb{R}\}.
\end{aligned}$$

s-in-G: s is a vertex in G.

source-val: $\mathit{dist}(s) = 0$.

general-source-val: $\mathit{dist}(s) \leq 0$.

From the previous two properties, we know $s \in V_f \cup V_{-\infty}$.

trian: For all $(u, v) \in E$, $\mathit{dist}(u) + c(u, v) \geq \mathit{dist}(v)$.

---

[4] Note that in this section negative cycles do not occur since we only consider graphs with nonnegative edge costs. In the next section however, they become relevant because we consider graphs with arbitrary edge costs.

In particular, if $(u, v) \in E$ and $u \in V_f$ then $v \in V_f \cup V_{-\infty}$. Thus, there are no edges in E from vertices in $V_f \cup V_{-\infty}$ to vertices in V_∞ and hence no vertex in V_∞ is reachable from s. An induction argument, see next section, yields $\mathit{dist}(v) \leq \mu(c, s, v)$ for all $v \in V$.

just: For all $v \in V_f$, if $v \neq s$, then there exists $(u, v) \in E$ such that $\mathit{dist}(v) = \mathit{dist}(u) + c(u, v) \wedge \mathit{enum}(v) = \mathit{enum}(u) + 1$.

Using the *just* property and the two properties that follow one can prove that $\mathit{dist}(v) \geq \mu(c, s, v)$ for all $v \in V_f$.

non-neg-cost: For all edges e in G, $c(e) \geq 0$ where c is the cost function.

Hence, $V_{-\infty} = \emptyset$ and $\mu(c, s, v) \neq -\infty$ for all $v \in V$.

no-path: For all $v \in V$, $\mathit{dist}(v) = \infty$ if and only if there is no path to v.

Therefore, $\mathit{dist}(v) = \infty$ if and only if $\mu(c, s, v) = \infty$ for all $v \in V$. From all of the above one can conclude that $\mathit{dist}(v) = \mu(c, s, v)$ for all $v \in V$. We give an overview of the Isabelle formal proof in the next section.

Proving the Witness Property

We present here the outline of the Isabelle/HOL proof of the witness property. 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* states that under the *shortest-path-non-neg-cost* locale assumptions, for any vertex v in G, $\mathit{dist}(v)$ is equal to the correct shortest path distance $\mu(c, s, v)$ from s to v using the cost function c.

Listing 2.2 shows our Isabelle locales. We separate the assumptions into three locales to avoid the use of unneeded assumptions when proving intermediate lemmas. This makes the intermediate lemmas more general, and hence, usable in other contexts. For example, we reuse some of the lemmas in this section for the verification of a checker for the more general shortest-path problem with arbitrary edge costs in explained in Section 2.2.3. The locale *basic-sp* subsumes the locale *fin-digraph* mentioned in Section 2.2.1. Moreover, it assumes it is given the function $\mathit{dist} : V \to (\mathbb{R} \cup \{\infty, -\infty\})$, an edge cost function $c : E \to \mathbb{R}$, and a start vertex s.

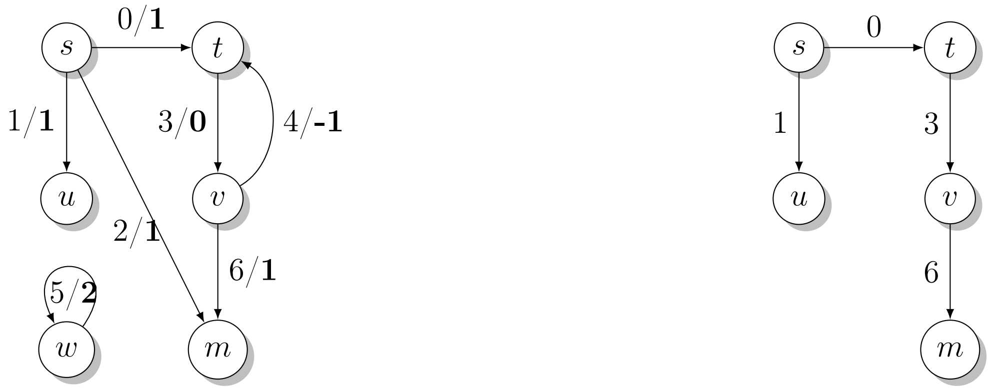
We split the proof of the witness property into two parts. First, we prove a lemma *dist-le-*μ using the locale *basic-sp*. The lemma states that $\mathit{dist}(v) \leq \mu(c, s, v)$ for every vertex $v \in V$. Then, we prove the lemma *dist-ge-*μ using the locale *basic-just-sp*. The lemma states that $\mathit{dist}(v) \geq \mu(c, s, v)$ for every vertex $v \in V$ under some extra assumptions (Listing 2.3). Later, we show that these extra assumptions hold in the locale *shortest-path-non-neg-cost*. Hence, we obtain a theorem stating that $\mathit{dist}(v) = \mu(c, s, v)$ for every $v \in V$ using the locale *shortest-path-non-neg-cost*.

```
locale basic-sp = fin-digraph +
  fixes dist :: α ⇒ ereal
  fixes c :: β ⇒ real
  fixes s :: α
  assumes general-source-val: dist s ≤ 0
  assumes trian: ⋀e. e ∈ arcs G ⟹ dist (head G e) ≤ dist (tail G e) + c e

locale basic-just-sp = basic-sp +
  fixes enum :: α ⇒ enat
  assumes just:
   ⋀v. v ∈ verts G ⟹ v ≠ s ⟹ enum v ≠ ∞ ⟹
     ∃ e ∈ arcs G.
       v = head G e ∧
       dist v = dist (tail G e) + c e  ∧
       enum v = enum (tail G e) + enat 1

locale shortest-path-non-neg-cost = basic-just-sp +
  assumes s-in-G: s ∈ verts G
  assumes source-val: dist s = 0
  assumes no-path: ⋀v. v ∈ verts G ⟹ (dist v = ∞ ⟷ enum v = ∞)
  assumes non-neg-cost: ⋀e. e ∈ arcs G ⟹ 0 ≤ c e
```

Listing 2.2. The *basic-sp* locale inherits from *fin-digraph* and additionally includes the triangle inequality assumption *trian*. The locale *basic-just-sp* inherits from *basic-sp* and additionally includes the justification assumption *just*.

```
lemma (in basic-just-sp) dist-ge-μ:
  fixes v :: α
  assumes v ∈ verts G
  assumes enum v ≠ ∞
  assumes dist v ≠ −∞
  assumes μ c s s = ereal 0
  assumes dist s = 0
  assumes ⋀u. u ∈ verts G ⟹ u ≠ s ⟹ enum u ≠ enat 0
  shows dist v ≥ μ c s v
```

Listing 2.3. The central lemma of the shortest-paths proof in Isabelle



(a) A directed graph G

(b) A reachability tree of G

vertex	s	t	u	v	m	w
parent-edge	$\bot$	0	1	3	6	$\bot$
num	0	1	1	2	3	$-$
dist	0	$-\infty$	1	$-\infty$	$-\infty$	∞
C	$=$	$\{(t$	$\rightarrow$	v	$\rightarrow$	$t)\}$

(c) Tree representation

Figure 2.3. 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 reachability tree of G that is rooted at start vertex $s \in V$. The function *dist* is the shortest path function and the tree is encoded by *parent-edge* and *num* according to the table in (c). Observe that vertex w is not reachable from s and that the cycle $t \rightarrow v \rightarrow t$ is a negative cycle.

2.2.3. Shortest Path with Arbitrary Edge Costs

The single-source shortest-paths problem (with arbitrary edge costs) for directed graphs is explained in detail in the LEDA book [Mehlhorn and Näher, 1999, Sections 6.6 and 7.5]. There they describe a pen-and-paper axiomatic characterization of the shortest path function. We give a characterization in terms of three functions

$$\begin{aligned} &\mathit{dist} : V \rightarrow \mathbb{R} \cup \{\infty, -\infty\}, \\ &\mathit{num} : V \rightarrow \mathbb{N}, \text{ and} \\ &\mathit{parent\text{-}edge} : V \rightarrow E \cup \{\bot\}. \end{aligned}$$

We assume that G and the functions satisfy the following witness properties. If all properties are satisfied, $\mathit{dist}(v) = \mu(c, s, v)$ for all $v \in V$. We again start by stating the properties using standard mathematical notation and then give the Isabelle formalization in the following section. Figure 2.3 provides an example.

We re-use the properties *fin-digraph*, *general-source-val*, *trian*, and *just* introduced in Section 2.2.2. The following properties make sure that *num* and *parent-edge* encode a tree rooted at s and containing all vertices in $V_f \cup V_{-\infty}$. In particular, every vertex in $V_f \cup V_{-\infty}$ is reachable from s.

s-assms: $s \in V$ and $num(s) = 0$.

pna: $v \in V_f \cup V_{-\infty}$ and $v \neq s$ implies $parent\text{-}edge(v) \neq \bot$, $num(v) = num(u)+1$, and $u \in V_f \cup V_{-\infty}$ where $parent\text{-}edge(v) = (u, v)$.

In this case study, we define $enum : V \to \mathbb{N} \cup \{\infty\}$ such that $enum(v) = \infty$ if $dist(v) \in \{\infty, -\infty\}$ and $enum(v) = num(v)$ otherwise. Along with *fin-digraph*, *general-source-val*, *trian*, and *just*, the next properties ensure the correctness of the *dist* function.

source-val: If $V_f \neq \emptyset$, then $dist(s) = 0$ [5].

Thus, if $V_f = \emptyset$ then $s \in V_{-\infty}$ and hence $dist(s) = -\infty$.

C-se: Let C be the set of negative cycles in G.

We define *pwalk* to be a function from vertices to paths. It is the path obtained by concatenating the edges defined by the *parent-edge* function from v to s for vertices in $V_f \cup V_{-\infty}$ different from s, otherwise it is the empty path.

int-neg-cyc: For each vertex $v \in V_{-\infty}$, $pwalk(v)$ intersects a cycle in C.

Hence, each vertex $v \in V_{-\infty}$ is connected to s with a walk that contains a negative cycle.

We first introduce the formalization and explain the proof that from the witness introduced above one can conclude that *dist* is the shortest path function, then we discuss why such a witness always exists since in this case it is not entirely straightforward.

Proving the Witness Property

This formalization builds on the Isabelle directed graphs library [Noschinski, 2014]. The theory file for this formalization is in the Isabelle archive of formal proofs [Rizkallah, 2013]. We formalize the axioms as assumptions in the *shortest-paths-neg-cyc* locale (as shown in Listing 2.4) and prove that under those locale assumptions *dist* is indeed the single-source shortest path function for directed graphs with arbitrary edge costs. As in the previous example, we separate the assumptions into several intermediate locales. This separation allows us to prove more general intermediate lemmas, with less assumptions, that could be used later on in other contexts. The Isabelle

---

[5] Note that this property is different to the *source-val* property in the previous section.

```
locale shortest-paths-init =
    fixes G :: (α, β) pre-digraph (structure)
    fixes s :: α
    fixes c :: β ⇒ real
    fixes num :: α ⇒ nat
    fixes parent-edge :: α ⇒ β option
    fixes dist :: α ⇒ ereal
    assumes graphG: fin-digraph G

locale shortest-paths-reachable =
    shortest-paths-init +
    assumes s-assms:
      s ∈ verts G
      num s = 0
    assumes pna:
      ⋀ v. ⟦ v ∈ verts G; v ≠ s; v ∉ V∞ ⟧ ⟹
        (∃ e ∈ arcs G. parent-edge v = Some e ∧
        head G e = v ∧ tail G e ∉ V∞ ∧
        num v = num (tail G e) + 1)

locale shortest-paths-basic =
  shortest-paths-reachable +
  basic-just-sp G dist c s enum +
  assumes source-val: (∃ v ∈ verts G. enum v ≠ ∞) ⟹ dist s = 0

locale shortest-paths-neg-cyc =
    shortest-paths-basic +
    fixes C :: (α × (β awalk)) set
    assumes C-se: C ⊆ {(u, p). dist u ≠ ∞ ∧ awalk u p u ∧ awalk-cost c p < 0}
    assumes int-neg-cyc: ⋀ v. v ∈ V−∞ ⟹ (fst ‘ C) ∩ pwalk-verts v ≠ {}
```

Listing 2.4. The *shortest-paths-init* locale inherits from *fin-digraph* and additionally defines more constants. The *shortest-paths-reachable* locale inherits from *shortest-paths-init* and additionally includes the assumptions *s-assms* and *pna*. The *shortest-paths-basic* locale inherits from both *shortest-paths-reachable* and *basic-just-sp* (see Listing 2.2). It additionally includes the *source-val* assumption. The final locale including all the witness assumptions is called *shortest-paths-neg-cyc*, in addition to the assumptions in *shortest-paths-basic* it includes the *C-se* and the *int-neg-cyc* assumptions.

shortest-paths-neg-cyc locale contains exactly the properties described above. The final theorem states that under the assumptions in *shortest-paths-neg-cyc*, for any vertex v in G, $\mathit{dist}(v)$ is equal to the correct shortest path distance $\mu(c, s, v)$ from s to v using the cost function c. The high level proof is split into three parts, for any vertex $v \in V$:

1. if $v \in V_\infty$ then $\mathit{dist}(v) = \mu(c, s, v) = \infty$,
2. if $v \in V_f$ then $\mathit{dist}(v) = \mu(c, s, v) \in \mathbb{R}$, and
3. if $v \in V_{-\infty}$ then $\mathit{dist}(v) = \mu(c, s, v) = -\infty$.

The first part follows directly from the lemma *dist-le-*μ proven in context of the locale *basic-sp*. The lemma states that for all vertices $v \in V$, $\mathit{dist}(v) \leq \mu(c, s, v)$. We start by proving that for any walk p from s to v, $\mathit{dist}(v)$ is less than or equal to the cost of p using cost function c. The proof follows by induction on the length of p using the *trian* and *general-source-val* assumptions. Hence $\mathit{dist}(v) \leq \mu(c, s, v)$ for all v by definition of $\mu(c, s, v)$.

The second part is proven using the lemma *dist-Vf-*μ in the context of the locale *shortest-paths-basic*. The lemma states that for all vertices $v \in V$ such that $\mathit{dist}(v) \in \mathbb{R}$, $\mathit{dist}(v) = \mu(c, s, v)$. Using the lemma *dist-le-*μ we already know that for all vertices $v \in V$, $\mathit{dist}(v) \leq \mu(c, s, v)$. It suffices to prove the lemma *dist-ge-*μ in the context of the locale *basic-just-sp* stating that for all vertices $v \in V$, $\mathit{dist}(v) \geq \mu(c, s, v)$ under the following extra assumptions: $\mathit{num}(v) \neq \infty$, $\mathit{dist}(v) \neq -\infty$, $\mu(c, s, s) = 0$, $\mathit{dist}(s) = 0$, and for all vertices v other than s, $\mathit{num}(v) \neq 0$ (see Listing 2.3). The first two assumptions follow directly from the fact that we consider vertices v for which $\mathit{dist}(v) \in \mathbb{R}$. The latter three assumptions are easily discharged in context of the locale *shortest-paths-basic*. For the vertex s if $\mathit{dist}(s) \in \mathbb{R}$ then $\mathit{dist}(v) = \mu(c, s, v) = 0$ using the *source-val* assumption and the trivial empty path from s to s. Moreover, the assumption stating that for all vertices v other than s, $\mathit{num}(v) \neq 0$ follows directly from *pna*.

Now we explain the proof of *dist-ge-*μ. The proof follows by induction on $\mathit{num}(v)$ for any vertex v. The base case is trivial. For the inductive case, using assumptions we know $v \neq s$. By the induction hypothesis and the lemma *dist-le-*μ we know $\mathit{dist}(u) = \mu(c, s, u)$. Using *just* we obtain a witnessing edge (u, v) such that $\mathit{dist}(v) = \mathit{dist}(u) + c(u, v) = \mu(c, s, u) + c(u, v)$. We obtain a path of cost $\mu(c, s, u) + c(u, v)$ from s to v by appending the edge (u, v) to the shortest path from s to u that has cost $\mu(c, s, u)$. Because such a path exists we know $\mu(c, s, v) \leq \mu(c, s, u) + c(u, v) = \mathit{dist}(v)$. By this we conclude our proof.

The third part is proven using the lemma *Vn-*μ*-ninf* in the context of the locale *shortest-paths-neg-cyc*. The lemma states that if a vertex $v \in V_{-\infty}$ then $\mu(c, s, v) = -\infty$. Using the *int-neg-cyc* and *C-se* assumptions we prove that there is a walk with a negative cycle from s to v. There is a theorem in the Isabelle graph library stating that if there is walk with a negative cycle between two vertices then the shortest path between them has cost $-\infty$. By this we conclude our proof and the proof that for all vertices *dist* is the shortest path function.

Existence of Witness

In this case study the existence of a witness that satisfies the witness property is not entirely obvious. Therefore, we shortly discuss why the witness exists for a correct shortest path function *dist* on a finite wellformed graph. For more information see [Mehlhorn and Näher, 1999].

Since the source vertex s is reachable from itself using the empty path, $s \in V_f \cup V_{-\infty}$. It is in $V_{-\infty}$ if there is a negative cycle passing through s and in this case V_f is empty. Hence the assumptions *s-assms*, *source-val* and *general-source-val* are true for a correct shortest path function *dist*.

Moreover, one can always construct a reachability tree from s to all reachable vertices v. The tree spans over all vertices in $V_f \cup V_{-\infty}$. We construct the reachability tree as follows. If $V_f = \emptyset$, we take any directed tree rooted at s and spanning over vertices in $V_{-\infty}$. If $V_f \neq \emptyset$, we start with a shortest path tree for vertices in V_f. Let $V'_{-\infty} \subseteq V_{-\infty}$ be the set of vertices lying on negative cycles and having an incoming edge from a vertex in V_f. For each vertex in $V'_{-\infty}$ we select one such edge and add the vertex and the edge to the reachability tree. In a third step, we expand the reachability tree to include all vertices in $V_{-\infty}$. As long as there is an edge (u, v) with $u \in V_{-\infty}$, u already part of the tree, and v not part of the tree, we add v and the edge (u, v) to the tree.

For each vertex v in $V_f \cup V_{-\infty} \setminus \{s\}$ let *parent-edge*(v) be the tree edge into v in the reachability tree and let *num*(v) be the depth of v in the reachability tree.

The *trian* assumption holds because we assume *dist* is the correct shortest path function, *just* holds because the reachability tree is a shortest path tree on V_f and hence we can take *parent-edge*(v) as the witnessing edge in *just*. Finally, taking C to be the set of negative cycles in G, *int-neg-cyc* holds by construction.

2.2.4. Maximum Cardinality Matching

Our last case study is about maximum cardinality matching in general graphs. Assume you run a service at a climbing gymnasium that matches rock climbing partners. You have climbers and each climber can be matched with a list of other climbers of the same climbing proficiency level. This situation is readily modeled as a graph. There is a vertex for each climber and an edge for each possible match. The goal of the service is to arrange a maximum number of matches so that as many climbers as possible can climb. 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

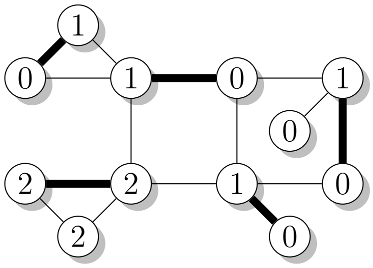


Figure 2.4. The vertex labels certify that the indicated matching is of maximum cardinality: All edges of the graph have either both endpoints labeled as 2 or at least one endpoint labeled as 1. Any matching can hence use at most one edge with both endpoints labeled 2 and at most four edges that have an endpoint labeled 1. Therefore, no matching has more than five edges. The matching shown consists of five edges (in bold).

share an endpoint. A matching has maximum cardinality if its cardinality is at least as large as that of any other matching. Figure 2.4 shows a graph, a maximum cardinality matching, and a witness of this fact. An *odd-set cover* L of a graph G is a labeling of the vertices of G with integers such that every edge of G is either incident to a vertex labeled 1 or connects two vertices labeled with the same number i and $i \geq 2$.

Theorem 2.2. *[Edmonds, 1965]. Let M be a matching in a graph G, and let L be an odd-set cover of G. For any $i \geq 0$, let n_i be the number of vertices labeled i. If*

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor, \tag{2.2}$$

then M is a maximum cardinality matching.

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

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

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

For every maximum cardinality matching M there is an odd-set cover L satisfying equality (2.2) [Mehlhorn and Näher, 1999, Section 7.7].; the proof of this is nontrivial and of no importance for the purpose of this work. The cover uses nonnegative vertex labels in the range 0 to $|V| - 1$ and all n_i's with $i \geq 2$ are odd. The *certifying algorithm for maximum cardinality matching* in LEDA returns a matching M and an odd-set cover L such that (2.2) holds. We instantiate our general approach as follows:

$$
\begin{aligned}
\text{input } x &= \text{an undirected graph } G\\
\text{output } y &= \text{a set of edges } M\\
\text{witness } w &= \text{a vertex labeling } L\\
\varphi(x) &= G \text{ and } M \text{ are wellformed and have no self-loops}\\
\mathcal{W}(x, y, w) &= M \text{ is a matching in } G,\ L \text{ is an odd-set cover for } G,\\
&\ \text{and Equation (2.2) holds}\\
\psi(x, y) &= M \text{ is a maximum cardinality matching in } G.
\end{aligned}
$$

Proving the Witness Property

We explain the Isabelle proof for the witness property, i.e., for Theorem 2.2. See Listing 2.5 for an excerpt of our formal Isabelle proof development. An older version of the Isabelle proof, that does not use the Isabelle graph library, can be found in [Rizkallah, 2011]. The formal proof follows the scheme of the textbook proof and is split into two main parts.

For $i \geq 2$, let M_i be the edges in M that connect two vertices labeled i, and let M_1 be the remaining edges in M. M_i is a set of edges. If we represent edges as sets, each with cardinality two, then M_i is a collection of sets. The sets M_i, $i \geq 1$, are disjoint. We use the definition of an odd-set cover to prove $M \subseteq \bigcup_{i \geq 1} M_i$, and thus $|M| \leq \sum_{i \geq 1} |M_i|$ by disjointness of the sets M_i. Let V_i be the vertices labeled i, and let $n_i = |V_i|$. We formally prove: $|M_1| \leq n_1$ and $|M_i| \leq \lfloor n_i/2 \rfloor$.

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

For $i \geq 2$, the proof of the inequality $|M_i| \leq \lfloor n_i/2 \rfloor$ is similar but more involved. We define the set of vertices V_i' to be $\bigcup_{i \geq 2} M_i$ and use the definition of an odd-set cover to prove $V_i' \subseteq V_i$. Since the edges in a matching are pairwise disjoint, we obtain $|V_i'| = 2 \cdot |M_i|$. Note also that $|V_i'|$ must be even since $|M_i|$ is a natural number. Thus, we can prove that $|M_i| \leq \lfloor |V_i'| \,/\, 2 \rfloor$, and hence, $|M_i| \leq \lfloor |V_i'| \,/\, 2 \rfloor \leq \lfloor |V_i| \,/\, 2 \rfloor = \lfloor n_i/2 \rfloor$.

2.3. Verification of Checker Implementations

We present two different methods for verifying the C implementations of the checkers. Section 2.3.1 explains work done by Eyad Alkassar and Sascha Böhme on verifying the checker correctness in VCC and linking the VCC and Isabelle/HOL formalizations. Back in 2011 when we started this work it was not tractable to verify C code within Isabelle/HOL using publicly available tools. We therefore decided to verify the checkers by using Isabelle/HOL as a backend to VCC.

```
type_synonym label = nat

definition disjoint_arcs :: (α, β) pre_graph ⇒ β ⇒ β ⇒ bool where
  disjoint_arcs G e₁ e₂ = (
     tail G e₁ ≠ tail G e₂ ∧ tail G e₁ ≠ head G e₂ ∧
     head G e₁ ≠ tail G e₂ ∧ head G e₁ ≠ head G e₂)

definition matching :: (α, β) pre_graph ⇒ β set ⇒ bool where
  matching G M = (
     M ⊆ arcs G ∧
     (∀e₁ ∈ M. ∀e₂ ∈ M. e₁ ≠ e₂ ⟶ disjoint_arcs G e₁ e₂))

definition OSC :: (α, β) pre_graph ⇒ (α ⇒ label) ⇒ bool where
  OSC G L = (
     ∀e ∈ arcs G.
        L (tail G e) = 1 ∨ L (head G e) = 1 ∨
        L (tail G e) = L (head G e) ∧ L (tail G e) ≥ 2)

definition weight :: label set ⇒ (label ⇒ nat) ⇒ nat where
  weight LV f = f 1 + ∑i ∈ LV. (f i) div 2

definition N :: α set ⇒ (α ⇒ label) ⇒ label ⇒ nat where
  N V L i = card {v ∈ V. L v = i}

locale matching-locale = digraph +
  fixes maxM :: β set
  fixes L :: α ⇒ label
  assumes matching: matching G maxM
  assumes OSC: OSC G L
  assumes weight: card maxM = weight {i ∈ L ‘ verts G. i > 1} (N (verts G) L)
```

Listing 2.5. Definitions and locale for the matching proof in Isabelle.

Our choice of VCC was encouraged by Alkassar's experience with VCC, the demonstration of VCC as a successful verification tool in the Verisoft XT project, and by Böhme's prior experience with using Isabelle as a backend to VCC in the Boogie verification condition generator. Here though our framework allows us to transfer cleaner chunks of mathematics to the Isabelle/HOL theorem prover and not overwhelm it with C code intricacies.

Meanwhile, several tools became available that enabled us to carry out the complete verification within Isabelle/HOL. Namely, the C-to-Isabelle parser developed for the seL4 project became open source and the AutoCorres tool that simplifies reasoning about code outputted by the parser emerged. In Section 2.3.3 we demonstrate that it is feasible to verify checkers entirely within Isabelle/HOL using those tools. Here we show this on the connected components example. This latter approach also proved to be successful in verifying the a graph non-planarity checker [Noschinski et al., 2014].

2.3.1. Verification of C code using VCC

This section explains work done by Eyad Alkassar and Sascha Böhme on verifying the checker correctness in VCC and linking the VCC and Isabelle/HOL formalizations. It is included in this thesis to give context to my contribution.

Export from VCC to Isabelle/HOL.

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

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

Listing 2.6. A representation of graphs in C. The field m gives the number of edges (and hence the length of the array es), and n gives the number of vertices in the graph.

functions to corresponding Isabelle terms.

Connected Components Checker

Implementation We begin by fixing a representation of graphs in the programming language C, as shown in Listing 2.6. Vertices are numbered consecutively from 0 to $\mathsf{n} - 1$. Edges are pairs where the first vertex is labeled s (for source), and the second vertex is labeled t (for target). Edges are stored in an array es, which is indexed by edge identifiers ranging from 0 to $\mathsf{m} - 1$. We require that the two vertices of each edge belong to the graph, i.e., that they are from the range $\{0, \ldots, \mathsf{n} - 1\}$, and call graphs with this property *wellformed*. We use the same data structure for directed and undirected graphs. For directed graphs, an edge e with $\mathsf{e.s} = \mathsf{u}$ and $\mathsf{e.t} = \mathsf{v}$ is directed from u to v. For undirected graphs, it represents the unordered pair {u,v}.

We represent spanning trees as explained previously in Section 2.2.1. Instead of functions, we use two arrays, parent_edge and num, in addition to a root vertex r. The parent_edge array maps r to a negative value, i.e., to a value that does not identify any edge.

The connected-graph checker is a function that accepts if the two functions check_r and check_parent_num (as shown in Listing 2.7) 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_edge[v] is incident to v and that the other endpoint of the edge has a number one smaller than num[v].

Checker Correctness To prove the two checker functions correct, we need to provide abstract representations for graphs and paths. We decided to keep them close to the concrete representation for two reasons. First, it makes detecting differences, and hence potential bugs, easier for the programmer. Second, it also makes reasoning for VCC simpler. The declaration of abstract graphs is given in Listing 2.8 together with the ghost predicate \wellformed for describing when an abstract graph is wellformed. This ghost predicate plays the role of the precondition φ in this case study. Our abstract version of the num array

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

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

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

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

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

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

Listing 2.7. The connected-components checker.

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

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

Listing 2.8. Abstract graphs and a predicate to describe wellformed graphs.

is a mapping from vertices to natural numbers. The abstract version of the parent_edge array is a mapping from vertices to the set $\mathbb{N} \cup \{\bot\}$; we use $\bot$ to model an undefined value. To represent this set, we define an algebraic datatype Option:

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

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

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

abstracts a concrete graph G into an abstract graph of type \Graph. Similarly abstraction functions \abs_parent_edge and \abs_num are defined to abstract tparent_edge and num respectively; we will refer to \abs_parent_edge(G, parent_edge) as P.

Using the abstract types, we define the witness predicate as a conjunction of two properties, one for each of the checker functions in Listing 2.7.

check_r: Vertex r is the root of the spanning tree:

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

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

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

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

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

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

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

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

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

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

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

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

The final part of the formal proof—linking the high-level proofs with the properties exported from VCC to Isabelle—is fairly straightforward. Proving that the precondition and the witness predicate (cf. Section 2.3.1) match the assumptions specified in the locale *connected-components-locale* involves no reasoning beyond syntactical rewriting. To instantiate these assumptions, we provide lifting functions that abstract from the concrete representations of graphs and spanning trees stemming from our VCC specification to the high-level representation used by the Isabelle graph library. Thus, if the checker accepts, the lifted high-level graph is connected. Establishing the checker postcondition (the connectivity of unlifted graphs) requires showing that any high-level path witnessing reachability between two vertices corresponds to an unlifted path. This is straightforward because our representation of paths in the VCC formalization is close to the path representation of the Isabelle graph library.

Shortest Path Checker

Implementation We adopt the data structures of the previous case study (Section 2.3.1) with the exception that the `num` array stores elements of type **int** instead of `Nat`. This is because vertices may now also be unreachable from the source vertex, and we encode this by requiring that `num` takes a negative value for such vertices. We represent distances from the source vertex to any other vertex by an array `dist` with elements of type **int**. Any negative value encodes ∞. Finally, the edge weights are modeled by an array `cost` that gives for every edge a value of type `ushort` (an abbreviation for **unsigned short**).

Based on these types, we implement the shortest-path checker as a function that accepts when all of the four functions given in Listing 2.9 accept. That is, we check that the source vertex `s` is indeed the starting point (in `check_start_val`), that the `dist` and `num` arrays are consistent with respect to unreachable vertices, i.e., either both are finite or both are infinite (in `check_no_path`), that the triangle inequality property (Section 2.2.2) is fulfilled (in `check_trian`), and that the parent edge of every vertex `v` defines its distance value (in `check_just`).

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

Note that there is an alternative approach where `parent_edge` is not part of the witness. In that case `check_just` has to be rewritten. When considering a node `v`, it has to iterate over all edges into `v` to find the edge that defines `dist[v]`. An efficient implementation of this iteration requires providing each vertex with the list of edges into it.

Checker Correctness We now define our abstract specification for the shortest-path checker. We use the same data structures as in the previous case study (Section 2.3.1) with the exception that the `num` mapping now takes vertices to extended naturals ($\mathbb{N} \cup \{\infty\}$), represented by the type `Enat`. Extended naturals provide an explicit value for infinity:

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

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

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

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

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

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

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

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

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

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

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

Listing 2.9. Functions composing the shortest-path checker. The predicate INF(x) abbreviates x < 0, and VAL(x) stands for the type cast (Nat)x; Nat is the C type **unsigned** as defined in Listing 2.6.

```
})
```

We define functions `\is_enat_inf` to check whether an extended natural is infinity and `\enat_val_of` to convert an extended natural distinct from infinity into the corresponding natural number. For better readability, we will write a $=_e \infty$ for `\is_enat_inf(a)`. Moreover, we provide the predicates `\enat_eq` (abbreviated by $=_e$) and `\enat_le` ($\leq_e$) to decide equality and less-or-equal of two extended naturals as well as a function `\enat_add` ($+_e$) for the sum of an extended natural and a natural number:

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

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

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

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

The preconditions of this case study are that G is a wellformed graph and that the source vertex s is a vertex of G, i.e., that s < G.num_verts holds. We formalize the witness predicate as a conjunction of four properties, one for each of the four checker functions in Listing 2.9.

check_start_val: Vertex s is indeed the starting point:

```
dist[s] $=_e$ \enat_val(0)
```

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

```
∀ \Vertex v; v < G.num_verts ⟶ (dist[v] $=_e \infty$ <--> num[v] $=_e \infty$)
```

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

```
∀ \Edge_Id i; i < G.num_edges ⟶
  dist[G.edge[i].trg] $\leq_e$ dist[G.edge[i].src] $+_e$ cost[i]
```

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

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

We have verified that each of these four properties holds if and only if the corresponding checker function accepts. The three functions check_no_path, check_trian, and check_just need additional annotations before VCC can verify their correctness. The loops in these functions have to be annotated with loop invariants that are, just as in the previous case study (Section 2.3.1), only simple variants of the postconditions above. Also, as for the connected-components checker, we need to explicitly state properties that are violated before every **return** FALSE statement. Such properties are reformulations of concrete properties on the abstract level. In addition, both check_trian and check_just require the graph under consideration to be wellformed, and check_just, furthermore, requires that num and dist are consistent (the postcondition of check_no_path). We add these requirements as preconditions to the checker functions.

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

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

Only particular instances of this datatype are paths in the given graph G. To qualify valid paths, we proceed in two steps. We first define a predicate that expresses the conditions under which a sequence of edges constitutes a walk in graph G from vertex u to vertex v (Listing 2.10). Second, we define a predicate to describe when the set of vertices of an edge sequence is distinct (Listing 2.11). A path from vertex u to vertex v in G is a walk p from u to v with distinct vertices. We define this as a predicate \is_path(G, p, u, v).

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

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

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

Listing 2.10. A walk from vertex u to vertex v is a finite sequence of connected edges of graph G where the source vertex of the first edge is u and the target vertex of the last edge is v.

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

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

Listing 2.11. Predicate \distinct_verts(G, p) holds if the set of vertices connected by path p is distinct. Predicate \occurs(G, u, v, p) is true if and only if u is either equal to v or equal to any vertex touched by path p.

We formally prove this property, under the assumption of the precondition and the witness predicate.

Linking this Isabelle proof with the specification exported from VCC is a matter of translating from one representation to another. We intentionally chose to define paths and their costs in VCC similar to the way they are defined in the Isabelle graph library to ease our translation proofs. Since there are several more concepts to relate than in the previous checker (Section 2.2.1), our proofs for the shortest-path checker are more tedious. Nevertheless, no complex reasoning is required. We establish that the assumptions of the *shortest-path-non-neg-cost* locale are implied by the checker precondition and witness predicate, and we prove that our final theorem proved in that locale implies the checker postcondition.

Maximum Cardinality Matching Checker

Implementation We build the checker using the graph data structure as in the previous case studies (Listing 2.6). We assume that graphs are wellformed and have neither self-loops nor duplicate edges. We treat the edges of a graph as undirected edges. Matchings are also represented by graphs. We require an additional witness in the form of an array `f` that maps edge identifiers of the matching to edge identifiers of the input graph. For instance, if a graph consists of three edges (identified as 0, 1 and 2) and the computed matching consists of the third edge (i.e., 2), then `f` would be an array with a single element 2 indicating how the only edge of the matching corresponds to the edges of the input graph. Finally, the vertex labeling is represented by an array `osc`, which is indexed by vertices and stores elements of type `Nat`. The checker function requires an auxiliary array `check` that can store as many elements of type `Nat` as there are vertices in the input graph, but at least two. We expect that this array is allocated elsewhere and given as input to the checker.

In addition to the checker function, there are four helper functions (Listing 2.12). The checker accepts if the first three of them accept and if the fourth function returns a value that is equal to the number of edges of the matching `M`. In short, the helper functions perform the following tasks. The function `check_subset` checks whether `M` is a subgraph of `G` with respect to the mapping `f`. The function `check_matching` checks that `M` is indeed a matching (contains no two edges that are incident). The function `check_osc` checks whether the vertex labeling is an odd-set cover and that vertex labels are in the range $\{0, \ldots, \mathsf{G{-}{>}n} - 1\}$. Finally, the function `weight` computes the sum on the right-hand side of Equation (2.2). This computation is optimized by first searching for the greatest vertex label, which can be considerably smaller than the maximal $\mathsf{G{-}{>}n} - 1$, and then summing up partial sums only until this greatest label. The main checker function passes the auxiliary array `check` to `check_matching` as the `degree_in_M` argument and to `weight` as the `count` argument.

```
bool check_subset(Graph∗ G, Graph∗ M, Nat∗ f)
{
   Edge_Id e;
   for (e = 0; e < M−>m; e++)
   {
      if (f[e] >= G−>m) return FALSE;
      if (M−>es[e].s == G−>es[f[e]].s && M−>es[e].t == G−>es[f[e]].t) continue;
      if (M−>es[e].s == G−>es[f[e]].t && M−>es[e].t == G−>es[f[e]].s) continue;
      return FALSE;
   }
   return TRUE;
}

bool check_matching(Graph∗ M, Nat∗ degree_in_M)
{
   Vertex v; Edge_Id e;
   for (v = 0; v < M−>n; v++) degree_in_M[v] = 0;
   for (e = 0; e < M−>m; e++)
   {
      if (degree_in_M[M−>es[e].s] == 1 || degree_in_M[M−>es[e].t] == 1) return FALSE;
      degree_in_M[M−>es[e].s] = 1;
      degree_in_M[M−>es[e].t] = 1;
   }
   return TRUE;
}

bool check_osc(Graph∗ G, Nat∗ osc)
{
   Edge_Id e; Vertex v, w;
   for (v = 0; v < G−>n; v++) if (osc[v] >= G−>n) return FALSE;
   for (e = 0; e < G−>m; e++)
   {
      v = G−>es[e].s;
      w = G−>es[e].t;
      if (osc[v] == 1 || osc[w] == 1 || (osc[v] == osc[w] && osc[v] ≥ 2)) continue;
      return FALSE;
   }
   return TRUE;
}

Nat weight(Graph∗ G, Nat∗ osc, Nat∗ count)
{
   Vertex v; Nat c, s, max = 1, r = (G−>n > 2) ? G−>n : 2;
   for (c = 0; c < r; c++) count[c] = 0;
   for (v = 0; v < G−>n; v++)
   {
      count[osc[v]] = count[osc[v]] + 1;
      if (osc[v] > max) max = osc[v];
   }
   s = count[1];
   for (c = 2; c < max + 1; c++) s += count[c] / 2;
   return s;
}
```

Listing 2.12. Maximum cardinality matching checker's helper functions.

Checker Correctness We build on the abstract graph data structure of Listing 2.8. We require that graphs are wellformed and contain no self-loops:

```
∀ \Edge_Id i; i < G.num_edges ⟶ G.edge[i].src ≠ G.edge[i].trg
```

nor duplicate edges:

```
∀ \Edge_Id i1, i2; i1 < G.num_edges ∧ i2 < G.num_edges ∧ i1 ≠ i2 ⟶
   G.edge[i1].src ≠ G.edge[i2].src ∨ G.edge[i1].trg ≠ G.edge[i2].trg
```

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

The witness predicate is a conjunction of four predicates, each related to one of the helper functions in Listing 2.12.

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

```
∀ \Edge_Id i; i < M.num_edges ⟶
   f[i] < G.num_edges ∧
   (M.edge[i].src = G.edge[f[i]].src ∧ M.edge[i].trg = G.edge[f[i]].trg ∨
    M.edge[i].src = G.edge[f[i]].trg ∧ M.edge[i].trg = G.edge[f[i]].src)
```

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

```
∀ \Edge_Id i1, i2;
   i1 < M.num_edges ∧ i2 < M.num_edges ∧ i1 ≠ i2 ⟶
   M.edge[i1].src ≠ M.edge[i2].src ∧ M.edge[i1].src ≠ M.edge[i2].trg ∧
   M.edge[i1].trg ≠ M.edge[i2].src ∧ M.edge[i1].trg ≠ M.edge[i2].trg
```

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

```
∀ \Edge_Id i; i < G.num_edges ⟶
   L[G.edge[i].src] = 1 ∨
   L[G.edge[i].trg] = 1 ∨
   L[G.edge[i].src] = L[G.edge[i].trg] ∧ L[G.edge[i].src] ≥ 2
```

weight: Equation (2.2) must hold. We define it stepwise. The number of vertices labeled with c is defined recursively:

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

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

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

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

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

That is, we have $n_1 + \sum_{i\geq 2}\lfloor n_i/2\rfloor$ = \full_weight(L, G.num_verts, m) with the same m as before. Finally, the predicate capturing Equation (2.2) is as follows:

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

Verifying the correctness of the checker (Section 2.3.1) is done in the same way as the earlier case studies for the first three predicates above. We only have to provide the right loop invariants, and simple variations of the predicates to be proved are sufficient. In check_matching, we need additional loop invariants. Along with the first loop, we accumulate the knowledge about the initialization of the degree_in_M array by specifying that the first positions of the array have already been set to 0:

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

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

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

Another invariant states that vertices, for which degree_in_M is still 0, cannot be part of any already checked edge:

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

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

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

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

Proving the weight function correct is the most intricate part of the checker verification. There are two properties that need to be shown: functional correctness and the absence of overflows. Functional correctness requires that the function computes the n_i and the overall sum of Equation (2.2) correctly, as specified by the weight predicate above. Absence of overflows requires that the additions in both the second and third loop do not overflow. Surprisingly, the absence of overflows is much harder to establish than functional correctness.

We concentrate first on functional correctness. The second loop updates the count array in a way that maintains the following property:

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

From this property follows this loop invariant on the third loop:

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

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

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

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

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

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

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

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

2.3.2. Verification of Imperative Simpl code

In the previous section, we described how to verify the implementation of C checkers using the automatic code verifier VCC. We verified the witness properties of the checkers in Isabelle/HOL and verified the implementation of the checkers in VCC. Using two system requires the added effort of duplicate formalization in both systems. In addition, it requires trusting a larger code base.

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 [Schirmer, 2006] 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.

In this section we describe the verification of a Simpl implementation of the connected components checker and the shortest path checker (with nonnegative edge costs) within Isabelle/HOL. This allows us to estimate how much of the verification effort is needed for the verification of the actual algorithm and how much is needed for dealing with C intricacies.

Connected Components Checker

Implementation and Checker Correctness We begin by fixing the types we use for the Simpl implementation (see Listing 2.13). The type *IGraph* represents a graph G by the number of vertices $\mathit{ivertex\text{-}cnt}\,G$, number of edges $\mathit{iedge\text{-}cnt}\,G$, and a function $\mathit{iedge\text{-}cnt}\,G$ mapping from edge IDs to edges. Vertices of G range over the set $\{0, \ldots, (\mathit{ivertex\text{-}cnt}\,G) - 1\}$. Edges IDs range over the set $\{0, \ldots, (\mathit{iedge\text{-}cnt}\,G) - 1\}$, and edges are pairs of vertices and are obtained using the function $\mathit{iedge\text{-}cnt}\,G$. A graph is *wellformed* if both endpoints are smaller than $\mathit{ivertex\text{-}cnt}\,G$.

Each of the conditions in Listing 2.1 is checked by a procedure. For example, the procedure *parent-num-assms* in Listing 2.13 checks *parent-num-assms* in the obvious way. The loop invariant *parent-num-assms-inv* states that *parent-num-assms* holds up to vertex i.

The keyword **VAR MEASURE** in the implementation (see Listing 2.13) introduces the measure function used for the termination proof and guides the automated tools in Isabelle to prove termination automatically. The command **ANNO** binds the logical variables that are to be used in the invariant.

Total correctness of each function is formulated as a Hoare triple; see Lemma *parent-num-assms-spec* in Fig. 2.13. Invoking the VCG and using the annotations (loop invariant and measure function) is sufficient for the correctness proof.

```
type_synonym IVertex = nat
type_synonym Edge-Id = nat
type_synonym IEdge = IVertex × IVertex
type_synonym IPEdge = IVertex ⇒ Edge-Id option
type_synonym INum = IVertex ⇒ nat
type_synonym IGraph = nat × nat × (Edge-Id ⇒ IEdge)

definition parent-num-assms-inv :: IGraph⇒IVertex⇒IPEdge⇒INum⇒nat⇒bool
where parent-num-assms-inv G r p n k ≡ ∀ i < k. i ≠ r → (case p i of None ⇒ False
  | Some x ⇒ x < iedge-cnt G ∧ snd (iedges G x) = i ∧ n i = n (fst (iedges G x)) + 1)

procedures parent-num-assms
  (G :: IGraph, r :: IVertex, parent-edge :: IPEdge, num :: INum | R :: bool)
where vertex :: IVertex, edge-id :: Edge-Id
in ANNO (G, r, p, n). ⦃ G = G ∧ r = r ∧ parent-edge = p ∧ num = n ⦄
  R := True ; vertex := 0 ;
  TRY
    WHILE vertex < ivertex-cnt G
    INV ⦃ R = parent-num-assms-inv G r parent-edge num vertex
      ∧  G = G ∧ r = r ∧ parent-edge = p ∧ num = n
      ∧ vertex ≤ ivertex-cnt G⦄
    VAR MEASURE (ivertex-cnt G − vertex)
    DO
      IF (vertex ≠ r) THEN
        IF parent-edge vertex = None THEN
          R := False ;
          THROW
        FI ;
        edge-id := the (parent-edge vertex) ;
        IF edge-id ≥ iedge-cnt G ∨ snd (iedges G edge-id) ≠ vertex ∨
          num vertex ≠ num (fst (iedges G edge-id)) + 1 THEN
          R := False ;
          THROW
        FI
      FI ;
      vertex := vertex + 1
    OD
  CATCH SKIP END
  ⦃G = G ∧ r = r ∧ parent-edge = p ∧ num = n
    ∧ R=parent-num-assms-inv G r parent-edge num (ivertex-cnt G)⦄

lemma (in parent-num-assms-impl) parent-num-assms-spec:
 ∀G r p n.  Γ⊢t⦃G = G ∧ r = r ∧ parent-edge = p ∧ num = n⦄
  R := PROC parent-num-assms(G, r, parent-edge, num)
  ⦃ R = parent-num-assms-inv G r p n (ivertex-cnt G)⦄
```

Listing 2.13. 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.

Shortest Path Checker

Implementation We use the same type for graphs used for the connected components checker. Similarly, for each of the properties assumed in the locale, we implement a procedure checking this property and returning *True* if and only if the property holds. For example, the annotated procedure *is-wellformed* in Listing 2.14 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.

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 2.14) 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 wellformed (which is one of the assumptions in the *shortest-path-non-neg-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.

2.3.3. Verification of C code within Isabelle/HOL

In this section, we describe the verification of a C implementation of the connected components checker within Isabelle/HOL. To translate from C to Isabelle we use the C-to-Isabelle parser that was developed as part of the seL4 project [Klein et al., 2010] 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 [Greenaway et al., 2012] 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 would be competitive. In the case studies we carried out, we found it to be competitive, if not superior.

Connected Components Checker

Implementation and Checker Correctness The C representation of graphs is similar to that in Simpl. In particular, numbers are now of bounded precision.

```
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 2.14. Excerpts from witness property, implementation, and verification of shortest paths in Isabelle/HOL.

This means we need to prove absence of overflow 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-edge* is represented as an array of (signed) **int**, and *num* as an array of **unsigned int**. We require as a precondition that the input graph is wellformed.

The *check-connected* checker is a function that accepts exactly when the two functions *check-r* and *check-parent-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-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 Listing 2.1. This proof happens under the assumption that the pointers to the graph, to its edges, to *num* and to *parent-edge* can be abstracted to Isabelle datatypes (using the *arrlist* predicate).

Experiences and Lessons Learned The successful verification of this checker encourages us that the AutoCorres approach is feasible. The effort for the verification of the C-version of the connectedness checker was similar to the effort required by the VCC approach. VCC knows more about C and this made it easier to reason about the C-program. This advantage of VCC would become more significant 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, complicating the VCC-approach. Formalizing a small number of graph-theoretic concepts sufficed because verifying that the C-checker correctly checks the assumptions from Figure 2.1 needs no graph-theoretic knowledge and hence there is a clear separation of labor between VCC and Isabelle/HOL. The disadvantage of double formalization becomes clearer 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 is an example to this effect [Noschinski et al., 2014]. 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 [Alkassar et al., 2014] 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 $\mathsf{num}\ (\mathit{fst}\ (\mathit{iedges}\ \mathsf{G}\ \mathsf{edge\text{-}id})) + 1$ (see Fig. 2.13) 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 2.1 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.

2.4. Related Work

Certifying Algorithms The notion of a certifying algorithm is ancient. In the 9th century, Al-Khawarizmi already described how to (partially) check the correctness of a multiplication in his book on algebra. The extended Euclidean algorithm for greatest common divisors is also certifying; it dates back to the 17th century. Yet, formal verification of checkers is recent.

Verifying Checkers In 1997, Bright et al. [Bright et al., 1997] verified a checker for a sorting algorithm that has been formalized in the Boyer-Moore theorem prover [Boyer and Moore, 1990]. De Nivelle and Piskac formally verified the checker for priority queues implemented in LEDA [de Nivelle and Piskac, 2005]. Bulwahn et al. [Bulwahn et al., 2008] describe a verified SAT checker, i.e., a checker for certificates of unsatisfiability produced by a SAT solver. They develop the checker and prove its correctness within Isabelle/HOL. Similar proof checkers have been formalized in the Coq [Bertot and Castéran, 2004] proof assistant [Darbari et al., 2010, Armand et al., 2010]. CeTA [Thiemann and Sternagel, 2009], a tool for certified termination analysis, is also based on formally verified checkers. In contrast to our approach, all mentioned checkers are entirely developed and verified within the language of a theorem prover. 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. [Besson et al., 2010] provides an example.

More on VCC In the Verisoft XT project [Verisoft XT, 2010] VCC was successfully used to verify tens of thousands of non-trivial C code. So far, the majority of its verification targets have been restricted to system-level code from the domain of microkernels and hypervisors [Baumann et al., 2009, Klein et al., 2010, Shi et al., 2012]. Our work extends the range of VCC applications to graph algorithms and, in general, to any code that requires nontrivial mathematical reasoning to establish full functional correctness.

Theorem Provers as Backends Previous work that proposes, as we do, the use of interactive theorem provers as backends to code verification systems comprises, for instance, the link between Boogie and Isabelle/HOL [Böhme et al., 2010] and the link between Why and Coq [Filliâtre and Marché, 2007]. Both systems have a C verifier frontend. Such approaches for connecting code verifiers and proof assistants usually give proof assistants the same information that is made available to the first-order engine, overwhelming the users of the proof assistants with a mass of detail. Instead, we allow only clean chunks of mathematics to move between the verifier and the proof assistant. This hides details of the underlying programming languages from the proof assistant, thus requiring the user to discharge only interesting proof obligations.

Verification within Theorem Provers Verifying imperative code within interactive theorem provers such as Coq, HOL [Gordon and Melham, 1993], or Isabelle/HOL is also an active field of research. It requires a formalization of the imperative language and its semantics within the theorem prover. Norrish presented a formal semantics of C formalized in the HOL theorem prover [Norrish, 1998]. Parallel to this work, a subset of C, called C0, was formalized in Isabelle/HOL [Leinenbach et al., 2005]. Schirmer developed a verification environment for sequential imperative programs within Isabelle and embedded C0 into this environment [Schirmer, 2006]; his verification environment is written in Simpl. Schirmer's work has been applied, for instance, to verify a compiler for C0 [Petrova, 2007]. Moreover, the Verisoft project [Alkassar et al., 2009] reasoned about Simpl code within Isabelle.

The seL4 microkernel that is written in low-level C was verified within Isabelle/HOL using the C-to-Isabelle parser [Klein et al., 2010]. The underlying approach is refinement starting from an abstract specification via an intermediate implementation in Haskell to the final C code. Coq [Bertot and Castéran, 2004] was used both for programming the CompCert compiler and for proving its correctness [Leroy, 2009]. CFML is a verification tool embedded in Coq that targets imperative Caml programs [Charguéraud, 2011]. It was used to verify several imperative data structures.

An Endless Fascination with Shortest Path Shortest-path (with nonnegative edge costs) algorithms, especially imperative implementations thereof, are popular as case studies for demonstrating code verification [Charguéraud, 2011, Nordhoff and Lammich, 2012, Böhme et al., 2008]. Existing verification efforts target full functional correctness as opposed to instance correctness. Verifying instance correctness is orthogonal to verifying the implementation of a particular shortest path algorithm. In particular, our work is directly applicable to any implementation of shortest-path that is instrumented to provide the necessary witness expected by our checker.

The Other Checkers To our knowledge, there has been no other attempt to verify algorithms or checkers for connected components or maximum cardinality matchings. We are also the first to attempt any verification work on the shortest path problem with arbitrary edge costs.

Code Generation Since checkers are fairly simple programs that are not performance critical, code generation is a viable alternative to our explored approaches. However, generating C programs from theorem provers is still beyond the state of the art. The next chapter describes recent related work, that is still in progress, where code generation is used to ease the Isabelle verification of a C file system.

3

Verification of a C File System

This chapter discusses an ongoing project at the Trustworthy Systems group at NICTA aiming to develop a framework for verifying file systems, in particular, it explains my contribution to the project. This project is in collaboration with Sidney Amani, Zillin Chen, Liam O'Connor, Gernot Heiser, Gabriele Keller, Gerwin Klein, Toby Murray and Yutaka Nagashima. I start by introducing the overall project and then explain my contribution.

The Trustworthy Systems group at NICTA aims to formally verify systems software. In the L4.verified project the group formally verified the seL4 microkernel [Klein et al., 2010]. An ongoing project aims to automatically synthesize device drivers from formal specifications [Ryzhyk et al., 2009].

File systems account for a large portion of the code base of a kernel. For example the Linux kernel source tree presently contains 49 different file systems. The file system code is a significant source of kernel bugs despite its conceptual simplicity. This is due to the frequent error handling. Most bugs in file systems are semantic faults, and hence static analysis, while useful [Ball et al., 2010, Bessey et al., 2010], is not sufficient for eliminating most bugs [Lu et al., 2014]. To ensure the reliability of file systems, formal verification of their full functional correctness is necessary [Keller et al., 2013].

Verification of file systems, however, is rather challenging due to their diversity and their large code bases. Their code though is conceptually straightforward making them good candidates for automation. The work proposes a framework for verifying full functional correctness of file systems in reasonable time through co-generation of code and proofs. One writes the file system in a non-Turing complete executable domain specific functional language, currently called CDSL. The type system of CDSL guarantees that all errors are handled. The CDSL compiler

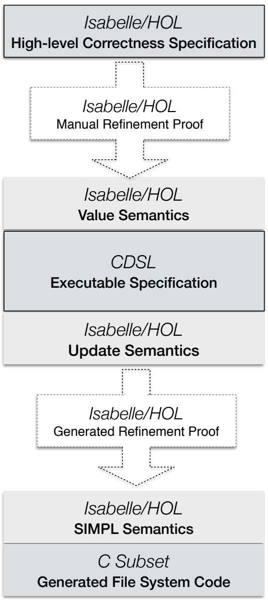


Figure 3.1. Structure of a CDSL correctness proof. Reprinted from [O'Connor-Davis et al., 2014]

generates an efficient C implementation and an executable Isabelle specification. The idea is to develop tools that generate the proof of correspondence between the C code and the generated executable specification automatically. This way instead of manually proving that a C program corresponds to its high level Isabelle specification, one only needs to manually prove that the executable and the high level Isabelle specification of the C code correspond, which significantly reduces the verification effort. See Figure 3.1 for more details.

CDSL is deeply embedded[1] into Isabelle/HOL and has two semantics that are formalized and proven equivalent in Isabelle/HOL. The first is the *value semantics* that provides a functional view of the language and does not talk about what is stored in the memory and pointer locations. Hence it is convenient to relate it to the high level specification. The second is the *update semantics* that provides a more imperative view and talks about the store and pointer locations and is therefore easier to relate to the C implementation. The language is still being extended to properly account for loops and arrays.

My contribution to the project is relating CDSL and C. I define a Hoare logic for the update semantics of CDSL and prove weakest precondition lemmas for CDSL statements. Moreover, I define a correspondence relation between CDSL and C statements and write lemmas relating CDSL statements with corresponding C statements. In order to relate return values in C and CDSL and states of C memory and CDSL stores, I define a correspondence between CDSL

---

[1] For a discussion of shallow and deep embeddings see Section 1.3.3.

values and C values. The state relation and return relation are different for every program depending on the structs used in the program. They therefore need to be automatically generated. The correspondence lemmas between statements need to be customized to those relations to allow for simplification. I manually proved that some small examples of CDSL and C code are equivalent. Those manual proofs need to eventually be automatically generated. This is work in progress.

3.1. CDSL

CDSL is a functional programming language with a linear type system. When an object has a linear type, this simply means that it is used exactly once. The functional semantics of the language eases verification and the linear type system allows many program properties to be ensured statically. Using linear types for all dynamic data structures maintains memory safety in the presence of destructive updates without using garbage collection [Wadler, 1990].

The *value semantics* provides a functional view, passing arguments by value. The *update semantics* is a more imperative view, with a mutable store, pointers, and destructive updates. Here we explain the update semantics because it is the relevant semantics for relating CDSL code to the C code generated from it. A formal description of both semantics and a formal proof of their equivalence are given in [O'Connor-Davis et al., 2014]. It is also proven statically that every CDSL program terminates.

3.1.1. Abstract Syntax

CDSL values have one of two types: *boxed* or *unboxed*. Values of unboxed type are passed by copy and values of boxed type are either linear or shareable.

A CDSL program either returns a list of *Success* values of type *Succeeds* or an error code and a list of *Failure* values of type *Fails*. The type *CanFail* indicates that a return value is either of type *Succeeds* or *Fails*.

The CDSL language is defined by a datatype statement with the following constructors:

Return vs : returns a list of expressions vs.

Fail e vs : fails and returns an error code e and a list of expressions vs.

Seq s t : Sequential composition of a statement s and a statement continuation t.

If c s t : Conditional statement, if c then s else t.

AApp $f\ es$: Application of an abstract function, defined outside of CDSL, called f on arguments es.

Case $e\ n\ s\ t$: Case statement with more than two arguments

Esac $e\ n\ s$: Case statement with only two arguments

LetBang $vs\ s\ t$: similar to Seq statements. In addition to the statement s and the statement continuation t, it also takes a list of linear objects vs and allows programmers to read the content of those objects more than once without altering them. Without LetBang statements one would have to replace a linear object with a new linear object every time it is read.

Take $r\ f$: A record is modeled as a list of (field name × value option) paris. Take $r\ f$ returns the value of the field f of a linear record r and replaces the value by *None*. The field cannot have value *None* before the operation.

Put $r\ f\ e$: Sticks e into field f of linear record r, The field must have value *None* before the operation.

Promote s : Generalizes the return type of statement s from *Succeeds* or *Fails* to *CanFail*.

For $i\ args\ accs\ reads\ a$: Loops restricted to iterators over data structures.

The typing relation *stmt-type* $\gamma\ s\ \tau$ states that statement s is well-typed[2] and has type τ under the type environment γ. The environment matching relation *uval-typed-pointers-env* $\sigma\ \Gamma\ \gamma$ *input ireads* matches a value environment Γ and a type environment γ. For more information about the type system and the environment matching relation, see [O'Connor-Davis et al., 2014].

3.1.2. Update Semantics

The *update* semantics resembles that of an imperative language: Values may also be *pointers* that are names of locations in the mutable store σ.

The big-step evaluation relation $\Gamma \vdash (\sigma, c) \Downarrow! (\sigma', r)$ states that under the value environment Γ, program c using store σ evaluates to value r and updates the store to σ'. The store maps each location to either *Some* value or the free space *None*. Variables are represented by De Bruijn indices [de Bruijn, 1972] rather than names. [3] The value environment Γ is a list of values (of the variables). It represents a map from variables to values where the first value is the value of the variable with De Bruijn index zero and so on.

---

[2] i.e., respects the type system of CDSL

[3] A De Bruijn index is a natural number representing the occurrence of a variable in a term. It denotes the number of intermediate binders between the occurrence of a variable and its corresponding binder.

We use the notation $[e]^{\sigma}_{\Gamma}$ to mean the value resulting from evaluating an expression e under the value environment Γ and the CDSL store σ. We overload the notation $[vs]^{\sigma}_{\Gamma}$ to refer to the list of values resulting from evaluating the list of expressions vs. The following is a selection of the rules defining the update semantics of CDSL:

$$\textsf{Return}\ \frac{}{\Gamma \vdash (\sigma, \textsf{Return}\ vs) \Downarrow! \ (\sigma, \mathit{Success}[vs]^{\sigma}_{\Gamma})}$$

$$\textsf{If}\ \frac{\Gamma \vdash (\sigma, \mathit{if}\ [b]^{\sigma}_{\Gamma} = \mathit{true\ then}\ s\ \mathit{else}\ t) \Downarrow! \ (\sigma', r)}{\Gamma \vdash (\sigma, \textsf{If}\ b\ s\ t) \Downarrow! \ (\sigma', r)}$$

The C in the **Seq** and **LetBang** rules is a continuation of the form *OnlySuccess* t, *OnlyError* h, or *HandleError* $t\ h$. We present the rules for the evaluation relation on continuations $\Gamma \vdash (\sigma'', r, C) \Downarrow! \ (\sigma', r')$ later on in the section.

$$\textsf{Seq}\ \frac{\Gamma \vdash (\sigma, s) \Downarrow! \ (\sigma'', r) \quad \Gamma \vdash (\sigma'', r, C) \Downarrow! \ (\sigma', r')}{\Gamma \vdash (\sigma, \textsf{Seq}\ s\ C) \Downarrow! \ (\sigma', r')}$$

$$\textsf{LetBang}\ \frac{\Gamma \vdash (\sigma, s) \Downarrow! \ (\sigma'', r) \quad \Gamma \vdash (\sigma'', r, C) \Downarrow! \ (\sigma', r')}{\Gamma \vdash (\sigma, \textsf{LetBang}\ vs\ s\ C) \Downarrow! \ (\sigma', r')}$$

$$\textsf{Take}\ \frac{[lv]^{\sigma}_{\Gamma} = \mathit{Ptr}\ l \qquad \sigma\ l = \mathit{Some}\ (\mathit{RecVal}\ r) \qquad (f, \mathit{Some}\ rv) \in \mathit{set}\ \ r}{\Gamma \vdash (\sigma, \textsf{Take}\ lv\ f) \Downarrow! \ (\sigma(l := \mathit{Some}\ (\mathit{RecVal}\ r')), \mathit{Success}\ [\mathit{Ptr}\ \ l, rv])}$$

where r' is the record resulting from replacing the value of f in r by *None*.

$$\textsf{Put}\ \frac{[lv]^{\sigma}_{\Gamma} = \mathit{Ptr}\ l \qquad \sigma\ l = \mathit{Some}\ (\mathit{RecVal}\ r) \qquad (f, \mathit{None}) \in \mathit{set}\ \ r}{\Gamma \vdash (\sigma, \textsf{Put}\ lv\ f\ \ e) \Downarrow! \ (\sigma(l := \mathit{Some}\ (\mathit{RecVal}\ r')), \mathit{Success}\ [\mathit{Ptr}\ \ l])}$$

where r' is the record resulting from replacing the value of f in r by $[e]^{\sigma}_{\Gamma}$.

$$\textsf{AApp}\ \frac{\mathit{afun\text{-}sem\text{-}upd}\ \mathit{fn}\ (\sigma, [es]^{\sigma}_{\Gamma})(\sigma', rs)}{\Gamma \vdash (\sigma, \textsf{AApp}\ \mathit{fn}\ es) \Downarrow! \ (\sigma', rs)}$$

where $\mathit{afun\text{-}sem\text{-}upd}\ \ \mathit{fn}\ \ (\sigma, [es]^{\sigma}_{\Gamma})(\sigma', rs)$ defines the behavior of the abstract function *fn*.

The following are the evaluation rules for continuations:

$$\textsf{Cont-Success}\ \frac{vs@\Gamma \vdash (\sigma, s) \Downarrow! \ (\sigma', r')}{\Gamma \vdash (\sigma, \mathit{Success}\ vs, \mathit{OnlySuccess}\ s) \Downarrow! \ (\sigma', r')}$$

$$\textsf{Cont-Failure}\ \frac{\mathit{LVal}(\mathit{W32}\ \ v)\#vs@\Gamma \vdash (\sigma, t) \Downarrow! \ (\sigma', r')}{\Gamma \vdash (\sigma, \mathit{Failure}\ v\ vs, \mathit{OnlyError}\ t) \Downarrow! \ (\sigma', r')}$$

where $\mathit{LVal}(\mathit{W32}\ v)$ is the literal value corresponding to error value v.

$$\textsf{Cont-HandleError-Success}\ \frac{vs @ \Gamma \vdash (\sigma, s) \Downarrow! (\sigma', r')}{\Gamma \vdash (\sigma, \mathit{Success}\ vs, \mathit{HandleError}\ s\ t) \Downarrow! (\sigma', r')}$$

$$\textsf{Cont-HandleError-Failure}\ \frac{\mathit{LVal}(\mathit{W32}\ v) \# vs @ \Gamma \vdash (\sigma, t) \Downarrow! (\sigma', r')}{\Gamma \vdash (\sigma, \mathit{Failure}\ v\ vs, \mathit{HandleError}\ s\ t) \Downarrow! (\sigma', r')}$$

Most of the rules are straightforward, except for the rules Take and Put. These two rules operate on pointers and destructively update the records they point to in the store. The mutable store here is quite abstract compared to the heap of bytes on an actual machine or even the typed heap of [Greenaway et al., 2012]. We bridge the remaining gap by an automatically generated refinement proof, which is discussed in the next section.

3.2. Correspondence between C and CDSL

This section explains my contribution to the project. This work was initiated and supervised by Toby Murray. It builds towards automating the proof that CDSL programs corresponds to the C code generated from them. More precisely, the CDSL compiler produces C code from CDSL programs. We aim to automatically generate proofs of correctness for this generated C code.

The correctness statement between C and CDSL code is a refinement theorem between the CDSL program's update semantics and the semantics of the generated C code; i.e., we are generating a new proof for each program. We chose using the update semantics for this proof, because it is closer to the semantics of C. Since the update semantics and the value semantics are proven equivalent [O'Connor-Davis et al., 2014], we can add another abstraction step on top of this C correctness statement. This gives us a CDSL program in value semantics that is connected by formal proof to its C implementation and using translation validation [Pnueli et al., 1998], eventually to the final binary [Sewell et al., 2013].

We are making use of AutoCorres that abstracts the C code into monadic C code (for more details see Section 1.3.4). The target of our refinement proof is no longer low-level C semantics, but rather the monadic C abstraction; which simplifies automation. In particular, we define a correspondence relation *corres* between CDSL statements and monadic C statements. To ease the proof automation process later on, we define a Hoare logic for the update semantics of CDSL and give weakest precondition lemmas that create a VCG for CDSL. The semantics of CDSL loops at the time of writing are in flux and so we consider for now only the loop free subset of CDSL.

The C code generation is straightforward and does not do any global optimizations or transformations. The code generation for each individual construct corresponds to precisely one *corres* proof rule in Isabelle that connects the CDSL update semantics for that construct with its monadic C representation. The refinement proof for the entire program then simply composes these rules appropriately. We already have some manual refinement proofs on small examples that show that the *corres* proof rules compose. The composition of the rules can be automated leading to an automatic proof generation of the correctness of the generated C. This is ongoing work.

The *corres* proof rules depend on preconditions about the expected state of the program, for instance, preconditions about the type and validity of pointers in the heap. We propagate the conditions similarly to the proof calculus of [Cock et al., 2008]. Since our *corres* proof rules are specialized to CDSL and to the operation of the compiler, we can predict the form of these preconditions and design proof rules to combine them. This is the basis for automating these proofs of refinement.

3.2.1. A Hoare Logic and Weakest Precondition Rules

Every type correct CDSL program terminates. Therefore we only need a Hoare logic for partial correctness. A Hoare formula is of the form $\{\!|P|\!\}\ c\ \{\!|R, E|\!\}$ and denotes that program c starting in a state satisfying P ensures R in the case of success and E in the case of failure. The following is the notion of validity of a Hoare formula:

definition
$\textit{cdsl-valid} :: \textit{uval environment} \Rightarrow (\textit{store} \Rightarrow \textit{bool}) \Rightarrow\ \textit{statement} \Rightarrow$
$(\textit{uval list} \Rightarrow \textit{store} \Rightarrow \textit{bool}) \Rightarrow (\textit{uval list} \Rightarrow \textit{store} \Rightarrow \textit{bool}) \Rightarrow \textit{bool}$
$(\_ \vdash \{\!|\_|\!\}\ \_\ \{\!|\_,\_|\!\})$
where
$\Gamma \vdash \{\!|P|\!\}\ c\ \{\!|R,E|\!\} \equiv \forall r\ \sigma\ \sigma'\ \gamma\ \tau\ .\ P\ \sigma \longrightarrow \textit{stmt-type}\ \gamma\ \ c\ \tau \longrightarrow \Gamma \vdash (\sigma,c)\ \Downarrow!\ (\sigma',r) \longrightarrow$
$(\text{case}\ r\ \text{of}\ \textit{Success}\ vs \Rightarrow R\ vs\ \sigma'\ |\ \textit{Failure}\ e\ vs \Rightarrow E\ ((\textit{val-of-error}\ e)\#vs)\ \sigma')$

Let c be a program that is well-typed and let σ be a start state that satisfies the precondition P. When c executes, it results in a return value r and a state σ'. If r is a successful return value *Success vs*, then vs and σ' satisfy the post condition R. Otherwise, if r is a failure return value *Failure e vs*, then e, vs, and σ' satisfy the post condition E. Note that the post conditions R and E take a list of values vs and a state σ' but the precondition only takes a state.

The Hoare logic rules are syntax directed and most of them are weakest precondition rules. This eases the automatic application of the rules for the purpose of proof automation. We prove the rules in Isabelle/HOL. The following are the Hoare logic rules:

$$\mathsf{Return}\ \frac{}{\Gamma \vdash \{\!|\lambda\sigma.R\ [vs]^{\sigma}_{\Gamma}\ \sigma|\!\}\mathsf{Return}\ vs\{\!|R, E|\!\}}$$

$$\textsf{Fail}\ \dfrac{}{\Gamma \vdash \{\!|\lambda\sigma.E\ ((\mathit{val\text{-}err\text{-}val}\ \sigma\ \Gamma\ ([e]^{\sigma}_{\Gamma}))\#[vs]^{\sigma}_{\Gamma})\ \sigma|\!\}\textsf{Fail}\ e\ vs\{\!|R,E|\!\}}$$

$$\textsf{If}\ \dfrac{\Gamma \vdash \{\!|Ps|\!\}s\{\!|R,E|\!\} \qquad \Gamma \vdash \{\!|Pt|\!\}t\{\!|R,E|\!\}}{\Gamma \vdash \{\!|\lambda\sigma.\mathit{if}\ [b]^{\sigma}_{\Gamma} = \mathit{true\ then\ Ps}\ \sigma\ \mathit{else\ Pt}\ \sigma|\!\}\ \textsf{If}\ b\ s\ t\ \{\!|R,E|\!\}}$$

We introduce three weakest precondition lemmas for Seq statements depending on the type of the return value of the first statement in the Seq. Note that in the first premise we extend the context Γ by some list r such that $R'\ r$ holds as a precondition of t. The r can be later instantiated to the list of return values of statement s.

$$\textsf{Seq1}\ \dfrac{\Gamma \vdash \{\!|P|\!\}s\{\!|R',E'|\!\} \qquad \forall r.(r@\Gamma) \vdash \{\!|R'\ r|\!\}\ t\ \{\!|R,E|\!\}}{\Gamma \vdash \{\!|P|\!\}\textsf{Seq}\ s\ (\mathit{OnlySuccess}\ t)\{\!|R,E|\!\}}$$

$$\textsf{Seq2}\ \dfrac{\Gamma \vdash \{\!|P|\!\}s\{\!|R',E'|\!\} \qquad \forall r.(r@\Gamma) \vdash \{\!|E'\ r|\!\}\ h\ \{\!|R,E|\!\}}{\Gamma \vdash \{\!|P|\!\}\textsf{Seq}\ s\ (\mathit{OnlyError}\ h)\{\!|R,E|\!\}}$$

$$\textsf{Seq3}\ \dfrac{\begin{array}{c}\Gamma \vdash \{\!|P|\!\}s\{\!|R',E'|\!\}\\ \forall r.(r@\Gamma) \vdash \{\!|R'\ r|\!\}\ t\ \{\!|R,E|\!\} \qquad \forall r.(r@\Gamma) \vdash \{\!|E'\ r|\!\}\ h\ \{\!|R,E|\!\}\end{array}}{\Gamma \vdash \{\!|P|\!\}\textsf{Seq}\ s\ (\mathit{HandleError}\ t\ h)\{\!|R,E|\!\}}$$

The difference between Seq and LetBang statements is related to the type system and not to how they evaluate. Therefore, weakest precondition lemmas for LetBang statements are very similar to those of Seq statements.

$$\textsf{LetBang1}\ \dfrac{\Gamma \vdash \{\!|P|\!\}s\{\!|R',E'|\!\} \qquad \forall r.(r@\Gamma) \vdash \{\!|R'\ r|\!\}\ t\ \{\!|R,E|\!\}}{\Gamma \vdash \{\!|P|\!\}\textsf{LetBang}\ vs\ s\ (\mathit{OnlySuccess}\ t)\{\!|R,E|\!\}}$$

$$\textsf{LetBang2}\ \dfrac{\Gamma \vdash \{\!|P|\!\}s\{\!|R',E'|\!\} \qquad \forall r.(r@\Gamma) \vdash \{\!|E'\ r|\!\}\ h\ \{\!|R,E|\!\}}{\Gamma \vdash \{\!|P|\!\}\textsf{LetBang}\ vs\ s\ (\mathit{OnlyError}\ h)\{\!|R,E|\!\}}$$

$$\textsf{LetBang3}\ \dfrac{\begin{array}{c}\Gamma \vdash \{\!|P|\!\}s\{\!|R',E'|\!\}\\ \forall r.(r@\Gamma) \vdash \{\!|R'\ r|\!\}\ t\ \{\!|R,E|\!\} \qquad \forall r.(r@\Gamma) \vdash \{\!|E'\ r|\!\}\ h\ \{\!|R,E|\!\}\end{array}}{\Gamma \vdash \{\!|P|\!\}\textsf{LetBang}\ vs\ s\ (\mathit{HandleError}\ t\ h)\{\!|R,E|\!\}}$$

$$\textsf{Take}\ \dfrac{}{\begin{array}{l}\Gamma \vdash \{\!|\lambda\sigma.\forall l\ r\ rv.[lv]^{\sigma}_{\Gamma} = \mathit{Ptr}\ l \longrightarrow \sigma\ l = \mathit{Some}(\mathit{RecVal}\ r) \longrightarrow\\ (f, \mathit{Some}\ rv) \in \mathit{set}\ r \longrightarrow R\ [\mathit{Ptr}\ l, rv](\sigma(l := \mathit{Some}\ (\mathit{RecVal}\ r')))|\!\}\\ \textsf{Take}\ lv\ f\{\!|R,E|\!\}\end{array}}$$

where r' is the record resulting from replacing the value of f in r by None.

$$\textsf{Put}\ \frac{}{\begin{array}{l}\Gamma \vdash \{\!|\lambda\sigma.\forall l\ r.[lv]^{\sigma}_{\Gamma} = Ptr\ l \longrightarrow \sigma\ l = Some(RecVal\ r) \longrightarrow \\ (f, None) \in set\ r \longrightarrow R\ [Ptr\ l](\sigma(l := Some\ (RecVal\ r'))) |\!\} \\ \textsf{Put}\ lv\ f\ e \{\!|R, E|\!\}\end{array}}$$

where r' is the record resulting from replacing the value of f in r by $[e]^{\sigma}_{\Gamma}$.

$$\textsf{Case}\ \frac{\begin{array}{c}\forall l.(Ptr\ l)\#\Gamma \vdash \{\!|Ps\ l|\!\}s\{\!|R, E|\!\} \\ \forall t\ l.t \neq t' \longrightarrow (VariantVal\ t\ l)\#\Gamma \vdash \{\!|Ps'\ t\ l|\!\}s'\{\!|R, E|\!\}\end{array}}{\begin{array}{c}\Gamma \vdash \{\!|\lambda\sigma.\forall t\ l.[vv]^{\sigma}_{\Gamma} = VariantVal\ t\ l \longrightarrow \\ (if\ t' = t\ then\ Ps\ l\ \sigma\ else\ Ps'\ t\ l\ \sigma)|\!\}\textsf{Case}\ vv\ t'\ s\ s'\{\!|R, E|\!\}\end{array}}$$

$$\textsf{Esac}\ \frac{\forall l.(Ptr\ l)\#\Gamma \vdash \{\!|Ps\ l|\!\}s\{\!|R, E|\!\}}{\begin{array}{c}\Gamma \vdash \{\!|\lambda\sigma.\forall t\ l.[vv]^{\sigma}_{\Gamma} = VariantVal\ t\ l \longrightarrow \\ (t' = t \wedge Ps\ l\ \sigma)|\!\}\textsf{Esac}\ vv\ t'\ s\{\!|R, E|\!\}\end{array}}$$

$$\textsf{AApp}\ \frac{}{\begin{array}{c}\Gamma \vdash \{\!|\lambda\sigma.\forall\sigma' rs.afun\text{-}sem\text{-}upd\ fn\ (\sigma, [es]^{\sigma}_{\Gamma})(\sigma', rs) \longrightarrow \\ (case\ rs\ of\ Success\ vs \Rightarrow R\ vs\ \sigma' \mid \\ Failure\ e\ vs \Rightarrow E\ ((val\text{-}of\text{-}error\ e)\#vs)\ \sigma')|\!\} \\ \textsf{AApp}\ fn\ es\{\!|R, E|\!\}\end{array}}$$

where $afun\text{-}sem\text{-}upd\ fn\ (\sigma, [es]^{\sigma}_{\Gamma})(\sigma', rs)$ defines the behavior of the abstract function *fn*.

3.2.2. State Relation and Return Value Relation

We introduce the value relation *val-rel* relating CDSL values and monadic values. The relation *val-rel* is used to define the state relation *srel* that relates the CDSL store to the monadic heaps. The return value relation *rrel* that relates return values is also defined using *val-rel*. The definition of *corres*, that appears in the next section, depends on the relations *srel* and *rrel*. Moreover, some of the *corres* lemmas, such as the *corres* lemma for conditional statements, also uses the definition of *val-rel*.

The relation *val-rel* is defined differently on values of different monadic types. For basic types such as words of different lengths *val-rel* is defined statically. The relation *val-rel* is defined on words as follows:

```
definition val-rel-word-def:
 val-rel (x :: α word) uv ≡
   (if size x = 64 then (case uv of LVal (W64 y) ⇒ ucast x = y | _ ⇒ False)
    else if size x = 32 then (case uv of LVal (W32 y) ⇒ ucast x = y | _ ⇒ False)
    else if size x = 16 then (case uv of LVal (W16 y) ⇒ ucast x = y | _ ⇒ False)
    else if size x =  8 then (case uv of LVal (W8 y)  ⇒ ucast x = y |
                                         LVal (Bl y) ⇒  (x≠0) = y | _ ⇒ False)
    else False)
```

where *LVal* (*W64* y) is the literal value of a CDSL word y of size 64, and similarly for words of other lengths. A CDSL boolean *Bl* y is represented in the monadic C semantics as a word of size 8.

A monadic struct and a CDSL record are related if there are values in the fields of the record that are related, using *val-rel*, to the values in the fields of the struct. CDSL arrays also correspond to monadic structs and relating them is work in progress.

Let σ be a CDSL store and h be a monadic heap. The heap relation *heap-rel* σ h holds if for every pointer p that points to a non-empty value u in σ the heap h is also valid at pointer p and points to a value v that is related to u using the value relation *val-rel*. We define the state relation *srel* as the set of pairs of CDSL stores σ and monadic heaps h that are related using the relation *heap-rel*, i.e., $\mathit{srel} = \{(\sigma, h).\ \mathit{heap\text{-}rel}\ \sigma\ h\}$.

A CDSL function either succeeds and returns a list of values or it fails and returns a list of values and an error code. A corresponding monadic function returns a struct that has a field indicating whether or not the function succeeded and additionally contains for every value in the list of CDSL returns a corresponding field containing a related value. The return value relation *rrel* relates a CDSL return list to a corresponding monadic return struct in the obvious way.

3.2.3. Correspondence Proof Rules

In this section, we present the *corres* proof rules that relate CDSL statements to C statements. The rules are proven in Isabelle/HOL. The C code is generated by the CDSL compiler and simplified by AutoCorres to monadic code; or more precisely, to a non-deterministic state monad without exceptions (for more details see Section 1.3.4).

The *corres* relation takes as input a CDSL program c, a monadic C program m, a state relation *srel* defining the relation between the CDSL store and the C heap, a return value relation *rrel* that states which values in the list returned by CDSL are related to which values in the struct returned by C. It also takes the CDSL value environment Γ, a precondition P on the CDSL state, and a precondition P' on the C state. The following is the formal definition of the *corres* relation:

definition

$$\begin{array}{l}
\quad \mathit{corres} :: (\mathit{store} \times {}'\mathrm{s})\ \mathit{set} \Rightarrow (\mathit{uval\ return\text{-}value} \Rightarrow \alpha \Rightarrow \mathit{bool}) \Rightarrow \\
\qquad\qquad (\mathit{store} \Rightarrow \mathit{bool}) \Rightarrow ({}'\mathrm{s} \Rightarrow \mathit{bool}) \Rightarrow \\
\qquad\qquad \mathit{statement} \Rightarrow ({}'\mathrm{s},\alpha)\ \mathit{nondet\text{-}monad} \Rightarrow \mathit{uval\ environment} \Rightarrow \mathit{bool}
\end{array}$$

where

$$\begin{array}{l}
\quad \mathit{corres\ srel\ rrel}\ P\ P'\ c\ m\ \ \Gamma \equiv \\
\qquad (\forall \sigma\ s.\ (\sigma,s) \in \mathit{srel} \wedge P\ \sigma \wedge P'\ s \wedge \\
\qquad\ \ (\exists \gamma\ \mathit{input\ ireads}.\ (\exists\ \tau.\ \mathit{stmt\text{-}type}\ \gamma\ c\ \tau) \wedge \\
\qquad\ \ \ \ \mathit{uval\text{-}typed\text{-}pointers\text{-}env}\ \sigma\ \Gamma\ \gamma\ \mathit{input\ ireads}) \longrightarrow \\
\qquad\qquad \neg\ \mathit{snd}\ (m\ s) \wedge (\forall s'\ a.\ (a,s') \in \mathit{fst}\ (m\ s) \longrightarrow \\
\qquad\qquad\quad (\exists \sigma'\ r.\ \Gamma \vdash (\sigma,c) \Downarrow!\ (\sigma',r) \wedge (\sigma',s') \in \mathit{srel} \wedge \mathit{rrel}\ r\ a)))
\end{array}$$

A CDSL program c corresponds to a monadic C program m if and only if for every two related start states σ and s that satisfy the respective preconditions, if the program c is well-typed under a valid typing environment then (1) the monadic program m terminates without an exception, (2) the resulting states after execution of c and m are related, and (3) the return values of c and m are also related. We present a selection of the *corres* proof rules.

The following is the *corres* rule for Return statements.

$$\textsf{Return}\ \frac{}{\mathit{corres}\ \mathit{srel}\ \mathit{rrel}\ (\lambda\sigma.\mathit{rrel}\ (\mathit{Success}\ [\mathit{vs}]^{\sigma}_{\Gamma})\ x)\ \top\ (\textsf{Return}\ \mathit{vs})\ (\mathit{return}\ x)\ \Gamma}$$

The rule is straightforward and just states that a CDSL return statement and a monadic return statement correspond if the return values are related using the return value relation *rrel*. The symbol $\top$ is the Isabelle shorthand for $\lambda x.\mathit{True}$. In this rule $\top$ refers to the precondition of the monadic program.

Next we present the *corres* rule for conditional statements.

$$\textsf{If}\ \frac{\begin{array}{c}\forall\Gamma.\mathit{corres}\ \mathit{srel}\ \mathit{rrel}\ (Q\ \Gamma)\ Q'\ a\ a'\ \Gamma\\ \forall\Gamma.\mathit{corres}\ \mathit{srel}\ \mathit{rrel}\ (R\ \Gamma)\ R'\ b\ b'\ \Gamma\\ \forall\Gamma\sigma s.(\sigma, s) \in \mathit{srel} \to S\ \Gamma\ \sigma \longrightarrow S'\ s \longrightarrow \mathit{val\text{-}rel}\ (c'\ s)\ [c]^{\sigma}_{\Gamma}\end{array}}{\begin{array}{c}\mathit{corres}\ \mathit{srel}\ \mathit{rrel}\\ (S\ \Gamma\ \mathit{and}\ (\lambda\sigma.[c]^{\sigma}_{\Gamma} = \mathit{true} \longrightarrow Q\ \Gamma\ \sigma)\ \mathit{and}\ (\lambda\sigma.[c]^{\sigma}_{\Gamma} = \mathit{false} \longrightarrow R\ \Gamma\ \sigma))\\ (S'\ \mathit{and}\ (\lambda s.(c's \neq 0) \longrightarrow Q'\ s)\ \mathit{and}\ (\lambda s.(c's = 0) \longrightarrow R'\ s))\\ (\textsf{If}\ c\ a\ b)\ (\mathit{condition}\ (\lambda s.c's \neq 0)\ a'\ b')\ \Gamma\end{array}}$$

Abstracting from details the *corres* rule for conditional statements reads as follows: The CDSL statement If $c\ a\ b$ and the monadic statement $\mathit{condition}\ (\lambda s.c'\ s \neq 0)\ a'\ b'$ are related if the values of the conditions c and c' are related, a and a' are related, and b and b' are related. The preconditions of the resulting *corres* statement are the preconditions of c and c' being related and in addition, depending on whether or not c and c' are true the preconditions of one of the inner statements being related. Note that the *true* and *false* in the rule are of type boolean literal values.

The following is the *corres* rule is for sequencing statements Seq $c\ (\mathit{OnlySuccess}\ t)$.

$$\textsf{Seq}\quad \dfrac{\begin{array}{c} \mathit{corres}\ \mathit{srel}\ (\lambda r\ rv.(r = \mathit{Success}\ vs) \land \mathit{rrel}'\ rv\ vs)\ Q\ Q'\ c\ c'\ \Gamma \\ \forall rv\ vs.\mathit{rrel}'\ rv\ vs \longrightarrow \mathit{corres}\ \mathit{srel}\ \mathit{rrel}(R\ vs\ rv)(R'\ rv)\ t\ (t'\ rv)\ (vs@\Gamma) \\ \forall rv.\Gamma \vdash \{\!|P|\!\}c\{\!|\lambda vs\sigma.(\mathit{rrel}'\ rv\ vs) \longrightarrow R\ vs\ rv\ \sigma, \lambda vs\ \sigma.\mathit{True}|\!\} \\ \{\!|P'|\!\}\ c'\ \{\!|R'|\!\} \end{array}}{\begin{array}{c} \mathit{corres}\ \mathit{srel}\ \mathit{rrel}\ (P\ \mathit{and}\ Q)\ (P'\ \mathit{and}\ Q') \\ (\textsf{Seq}\ c\ (\mathit{OnlySuccess}\ t))\ (\mathit{do}\ rv \leftarrow c';\ (t'\ rv)\ \mathit{od})\ \Gamma \end{array}}$$

The rule roughly states that the CDSL statement Seq $c\ (\mathit{OnlySuccess}\ t))$ and the monadic statement $\mathit{do}\ rv \leftarrow c';\ (t'\ rv)\ \mathit{od}$ are related if c and c' are related, they return the related values $\mathit{Success}\ vs$ and rv respectively, and they have postconditions R and R' respectively, in addition, under the returns and postconditions of c and c' the statements t and t' are also related. Note that even though different Hoare calculi are used for CDSL statements (see Section 3.2.1) and monadic statements (see [Greenaway et al., 2012]), here we use the same Hoare notation.

The following is the *corres* rule for Take statements. It is less common as Take is a less common language constructor, however, as for all the other *corres* rules, it is just derived from the update semantics rule for the corresponding statement.

$$\textsf{Take}\quad \dfrac{}{\begin{array}{c} \mathit{corres}\ \mathit{srel}\ \mathit{rrel}\ (\lambda\sigma.[lv]^{\sigma}_{\Gamma} = \mathit{Ptr}\ \ p' \land (\forall r\ rv.\sigma\ p' = \mathit{Some}\ \ (\mathit{RecVal}\ \ r) \\ \longrightarrow (f, \mathit{Some}\ \ rv) \in \mathit{set}\ \ r \longrightarrow (\forall s.(\sigma, s) \in \mathit{srel} \longrightarrow \mathit{is\text{-}valid}\ s\ p' \land \\ (\sigma(p' \mapsto \mathit{RecVal}\ r'), s) \in \mathit{srel} \land \mathit{rrel}(\mathit{Success}[\mathit{Ptr}\ p', rv])(f's)))) \\ \top\ (\textsf{Take}\ lv\ f)\ (\mathit{do}\ y \leftarrow \mathit{guard}\ (\lambda s.\mathit{is\text{-}valid}\ s\ p');\mathit{gets}\ f'\ \mathit{od})\ \Gamma \end{array}}$$

where r' is the record resulting from replacing the value of f in r by *None*.

3.2.4. Related Work

File System Verification The verification of file systems has received some attention, as they are a well known source of system errors. Previous manual attempts [Arkoudas et al., 2004, Damchoom and Butler, 2009, Hesselink and Lali, 2012, Schierl et al., 2009] to provide verified file systems have, however, only proven the equivalence between two or more high-level specifications. None of these efforts relate the specification to an implementation of a realistic file system. These attempts also suffered from the overwhelming size and complexity of file system implementations. In order to prove complex properties about the file system, this previous research had to introduce serious limitations resulting in oversimplified filesystems that demonstrate the verification principle, but would not be usable in practice. There is also parallel ongoing work on the verification of a file system implementation [Schellhorn et al., 2014].

CDSL The High-Assurance Systems Programming (HASP)[(HASP), 2010] project shares our goals of improving the reliability of systems software. It is also similar in spirit, in that they seek to make these improvements by employing formal methods as well as programming language research. HASP's systems programming language, Habit, is similar to CDSL: a domain specific functional language. Providing a full formalization of Habit's semantics to facilitate formal reasoning and verification is one of the priorities of the project. [McCreight et al., 2010] show the correctness of a garbage collector in this project; however, to our knowledge, there exist no full formal language semantics yet. Habit is more general than CDSL. For example, it offers support for bit level and memory based data description, which we moved to a separate domain-specific language, DDSL.

The language PacLang [Ennals et al., 2004] is a domain-specific language that uses linear types to guide optimization of packet processing applications on network processors. The use of linear types in other areas of systems programming suggests that CDSL may find uses outside of file systems. Indeed, while PacLang is an imperative language, most PacLang programs could be translated to CDSL with little difficulty. The use of linear types in PacLang is purely designed for optimization, not for verification, and thus its type system is much less expressive than CDSL.

To the best of our knowledge, [Hofmann, 2000] is the only work which attempts to prove the equivalence of the functional and imperative interpretation of a language with a linear type system. The paper introduces a first order functional language with linear types, not unlike CDSL, and formalizes its semantics by denotation to set theory. It presents a translation of this language into C, and provides an informal proof of equivalence between the set theoretic interpretation and the C program. It is, however, not a rigorous mechanized formalization, and the approach would be unsuitable for machine-checked verification.

Correspondence Our work on creating a Hoare logic and proving correspondence lemmas between CDSL and monadic code is similar to the initial phase applied by AutoCorres that similarly automatically proves a correspondence between C and monadic code [Winwood et al., 2009, Greenaway et al., 2012]. Schirmer also created a Hoare logic and a VCG for Simpl [Schirmer, 2006].

4

Conclusion

The LEDA project [Mehlhorn and Näher, 1999] 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 [McConnell et al., 2011].

We described a framework for the verification of certifying computations and applied it to three nontrivial combinatorial problems: connectivity of graphs, shortest paths in graphs, and maximum cardinality matchings in graphs [Alkassar et al., 2011a, Alkassar et al., 2014]. Our work greatly increases the trustworthiness of certifying algorithms.

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

Our methodology can be applied to any other problem for which a certifying algorithm is known; see [McConnell et al., 2011] for a survey. We also prove the witness property of a checker for shortest paths with arbitrary edge costs [Rizkallah, 2013].

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

We then explored an alternative to the VCC approach which provides higher trust guarantees; namely, carrying out the complete verification within Isabelle/HOL [Noschinski et al., 2014]. 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 [Greenaway et al., 2012] promised to greatly simplify reasoning about C in Isabelle. We reworked the verification of the Simpl checkers for connectivity of graphs [Noschinski et al., 2014] and shortest paths [Rizkallah, 2014] within Isabelle. We also reworked the verification of the C checker for connectivity within Isabelle using the AutoCorres toolset [Greenaway et al., 2012, Greenaway et al., 2014]. The new AutoCorres approach was also used to verify the checker for graph non-planarity [Noschinski et al., 2014]; the non-planarity checker is amongst the most complex checkers in LEDA. Our experience with AutoCorres is positive. The AutoCorres approach 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). We concluded that the verification effort using this approach is comparable if not less than that of the previous approach.

The implementation of each of the advanced algorithms in LEDA took several man-months (recollection of Kurt Mehlhorn). 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 goes down with increased experience and development of the tools (cf. [Greenaway et al., 2014, Noschinski et al., 2014]). Our work demonstrates that the development of libraries of certifying programs with formally verified checkers is feasible at reasonable cost.

We explore the idea of verified code generation in Chapter 3. We describe a framework that makes the complete automatic generation of the code of a realistic C file system and the automated verification of the correctness of its modules feasible. The framework is based on the idea of co-generation of code and proofs from a domain specific language called CDSL. We define a Hoare logic for CDSL and prove correspondence lemmas between CDSL and C.

Bibliography

[Alkassar et al., 2011a] Alkassar, E., Böhme, S., Mehlhorn, K., and Rizkallah, C. (2011a). Verification of certifying computations. In *CAV*, pages 67–82.

[Alkassar et al., 2014] Alkassar, E., Böhme, S., Mehlhorn, K., and Rizkallah, C. (2014). A framework for the verification of certifying computations. *Journal of Automated Reasoning*, 52(3):241–273.

[Alkassar et al., 2011b] Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C., and Schweitzer, P. (2011b). An introduction to certifying algorithms. *it - Information Technology*, 53(6):287–293.

[Alkassar et al., 2009] Alkassar, E., Hillebrand, M. A., Leinenbach, D. C., Schirmer, N. W., Starostin, A., and Tsyban, A. (2009). Balancing the load: Leveraging semantics stack for systems verification. *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*, 42, Numbers 2-4:389–454.

[Arkoudas and Rinard, 2005] Arkoudas, K. and Rinard, M. C. (2005). Deductive runtime certification. *Electronic Notes in Theoretical Computer Science*, 113:45–63.

[Arkoudas et al., 2004] Arkoudas, K., Zee, K., Kuncak, V., and Rinard, M. (2004). Verifying a file system implementation. In Davies, J., Schulte, W., and Barnett, M., editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390. Springer Berlin Heidelberg.

[Armand et al., 2010] Armand, M., Grégoire, B., Spiwack, A., and Théry, L. (2010). Extending Coq with imperative features and its application to SAT verification. In Kaufmann, M. and Paulson, L., editors, *ITP*, volume 6172 of *LNCS*, pages 83–98. Springer.

[Ball et al., 2010] Ball, T., Bounimova, E., Kumar, R., and Levin, V. (2010). Slam2: Static driver verification with under 4% false alarms. In *FMCAD*, pages 35–42.

[Barnett et al., 2006] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387.

[Baumann et al., 2009] Baumann, C., Beckert, B., Blasum, H., and Bormer, T. (2009). Formal verification of a microkernel used in dependable software systems. In Buth, B., Rabe, G., and Seyfarth, T., editors, *Computer Safety, Reliability, and Security*, volume 5775 of *LNCS*, pages 187–200. Springer.

[Bertot and Castéran, 2004] Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Springer.

[Bessey et al., 2010] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C.-H., Kamsky, A., McPeak, S., and Engler, D. R. (2010). A few billion lines of code later: using static analysis to find bugs in the real world. In *Commun. ACM*, pages 66–75.

[Besson et al., 2010] Besson, F., Jensen, T. P., Pichardie, D., and Turpin, T. (2010). Certified result checking for polyhedral analysis of bytecode programs. In *TGC*, pages 253–267.

[Blum and Kannan, 1989] Blum, M. and Kannan, S. (1989). Designing programs that check their work. In *STOC*, pages 86–97.

[Böhme et al., 2008] Böhme, S., Leino, K. R. M., and Wolff, B. (2008). HOL-Boogie—An interactive prover for the Boogie program-verifier. In Mohamed, O. A., Muñoz, C., and Tahar, S., editors, *TPHOLs*, volume 5170 of *LNCS*, pages 150–166.

[Böhme et al., 2010] Böhme, S., Moskal, M., Schulte, W., and Wolff, B. (2010). HOL-Boogie—an interactive prover-backend for the Verifying C Compiler. *JAR*, 44(1–2):111–144.

[Boyer and Moore, 1990] Boyer, R. S. and Moore, J. S. (1990). A theorem prover for a computational logic. In *Conference on Automated Deduction*, volume 449 of *LNCS*, pages 1–15.

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

[Bulwahn et al., 2008] Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., and Matthews, J. (2008). Imperative functional programming with Isabelle/HOL. In Mohamed, O. A., Muñoz, C., and Tahar, S., editors, *TPHOLs*, volume 5170 of *LNCS*, pages 134–149.

[Charguéraud, 2011] Charguéraud, A. (2011). Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430.

[Cock et al., 2008] Cock, D., Klein, G., and Sewell, T. (2008). Secure microkernels, state monads and scalable refinement. In *TPHOLs*, volume 5170 of *LNCS*, pages 167–182.

[Cohen et al., 2009] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. (2009). VCC: A practical system for verifying concurrent C. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer.

[Damchoom and Butler, 2009] Damchoom, K. and Butler, M. J. (2009). Applying event and machine decomposition to a flash-based filestore in event-b. In Oliveira, M. V. M. and Woodcock, J., editors, *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, volume 5902 of *Lecture Notes in Computer Science*, pages 134–152. Springer.

[Darbari et al., 2010] Darbari, A., Fischer, B., and Marques-Silva, J. (2010). Industrial-strength certified SAT solving through verified SAT proof checking. In Cavalcanti, A., Deharbe, D., Gaudel, M.-C., and Woodcock, J., editors, *Theoretical Aspects of Computing*, volume 6255 of *LNCS*, pages 260–274. Springer.

[de Bruijn, 1972] de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the Church-Rosser theorem. *Indagationes Mathematicae (Koninglijke Nederlandse Akademie van Wetenschappen)*, 34(5):381–392. http://www.win.tue.nl/automath/archive/pdf/aut029.pdfElectronic Edition.

[de Moura and Bjørner, 2008] de Moura, L. M. and Bjørner, N. (2008). Z3: An efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer.

[de Nivelle and Piskac, 2005] de Nivelle, H. and Piskac, R. (2005). Verification of an off-line checker for priority queues. In *Software Engineering and Formal Methods*, pages 210–219. IEEE Computer Society.

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

[Ennals et al., 2004] Ennals, R., Sharp, R., and Mycroft, A. (2004). Linear types for packet processing. In Schmidt, D., editor, *13th ESOP*, volume 2986 of *LNCS*, pages 204–218. Springer.

[Filliâtre and Marché, 2007] Filliâtre, J.-C. and Marché, C. (2007). The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 173–177.

[Gordon et al., 1979] Gordon, M., Milner, R., and Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*.

[Gordon and Melham, 1993] Gordon, M. J. C. and Melham, T. F., editors (1993). *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press.

[Greenaway et al., 2012] Greenaway, D., Andronick, J., and Klein, G. (2012). Bridging the gap: Automatic verified abstraction of C. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 99–115.

[Greenaway et al., 2014] Greenaway, D., Lim, J., Andronick, J., and Klein, G. (2014). Don't sweat the small stuff: formal verification of C code without the pain. In

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014, page 45.

[(HASP), 2010] (HASP), T. H. A. S. P. P. (2010). The Habit programming language: The revised preliminary report.

[Hesselink and Lali, 2012] Hesselink, W. and Lali, M. (2012). Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44.

[Hofmann, 2000] Hofmann, M. (2000). A type system for bounded space and functional in-place update–extended abstract. In Smolka, G., editor, *ESOP*, volume 1782 of *LNCS*, pages 165–179. Springer.

[Keller et al., 2013] Keller, G., Murray, T., Amani, S., O'Connor-Davis, L., Chen, Z., Ryzhyk, L., Klein, G., and Heiser, G. (2013). File systems deserve verification too! In *Workshop on Programming Languages and Operating Systems (PLOS)*, pages 1–7, Farmington, Pennsylvania, USA.

[Klein et al., 2010] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2010). sel4: formal verification of an operating-system kernel. *CACM*, 53(6):107–115.

[Leinenbach et al., 2005] Leinenbach, D., Paul, W. J., and Petrova, E. (2005). Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In *SEFM*, pages 2–12. IEEE Computer Society.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *CACM*, 52(7):107–115.

[Lu et al., 2014] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Lu, S. (2014). A study of linux file system evolution. *TOS*, 10(1):3.

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

[McCreight et al., 2010] McCreight, A., Chevalier, T., and Tolmach, A. (2010). A certified framework for compiling and executing garbage-collected languages. In *15th ICFP*, pages 273–284. ACM.

[Mehlhorn and Näher, 1999] Mehlhorn, K. and Näher, S. (1999). *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press.

[Myreen, 2012] Myreen, M. O. (2012). Functional programs: Conversions between deep and shallow embeddings. In Beringer, L. and Felty, A. P., editors, *ITP*, volume 7406 of *Lecture Notes in Computer Science*, pages 412–417. Springer.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*.

[Nordhoff and Lammich, 2012] Nordhoff, B. and Lammich, P. (2012). Dijkstra's shortest path algorithm. *Archive of Formal Proofs*. http://afp.sourceforge.net/entries/Dijkstra_Shortest_Path.shtml, Formal proof development.

[Norrish, 1998] Norrish, M. (1998). *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge.

[Norrish, 2012] Norrish, M. (2012). C-to-Isabelle parser, version 0.7.2.

[Noschinski, 2014] Noschinski, L. (2014). A graph library for isabelle. *Mathematics in Computer Science*, pages 1–17.

[Noschinski et al., 2014] Noschinski, L., Rizkallah, C., and Mehlhorn, K. (2014). Verification of certifying computations through AutoCorres and Simpl. In *NASA Formal Methods*, pages 46–61.

[O'Connor-Davis et al., 2014] O'Connor-Davis, L., Keller, G., Amani, S., Murray, T., Klein, G., Chen, Z., and Rizkallah, C. (2014). CDSL version 1: Simplifying verification with linear types. Technical report, NICTA, Sydney, Australia.

[Petrova, 2007] Petrova, E. (2007). *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Saarbrücken.

[Pnueli et al., 1998] Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK. Springer-Verlag.

[Rizkallah, 2011] Rizkallah, C. (2011). Maximum cardinality matching. *Archive of Formal Proofs*. http://afp.sourceforge.net/entries/Max-Card-Matching.shtml, Formal proof development.

[Rizkallah, 2013] Rizkallah, C. (2013). An axiomatic characterization of the single-source shortest path problem. *Archive of Formal Proofs*. http://afp.sf.net/entries/ShortestPath.shtml, Formal proof development.

[Rizkallah, 2014] Rizkallah, C. (2014). A Simpl shortest path checker verification. In *Proceedings of Isabelle Workshop 2014*.

[Rizkallah, 2015] Rizkallah, C. (2015). Verification of certifying computations. http://dx.doi.org/10.5281/zenodo.16805.

[Ryzhyk et al., 2009] Ryzhyk, L., Chubb, P., Kuz, I., Sueur, E. L., and Heiser, G. (2009). Automatic device driver synthesis with termite. In *SOSP*, pages 73–86.

[Schellhorn et al., 2014] Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., and Reif, W. (2014). Development of a verified flash file system. In Ameur, Y. A. and Schewe, K., editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 9–24. Springer.

[Schierl et al., 2009] Schierl, A., Schellhorn, G., Haneberg, D., and Reif, W. (2009). Abstract specification of the ubifs file system for flash memory. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 190–206, Berlin, Heidelberg. Springer-Verlag.

[Schirmer, 2006] Schirmer, N. (2006). *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München.

[Sewell et al., 2013] Sewell, T. A. L., Myreen, M. O., and Klein, G. (2013). Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482.

[Shi et al., 2012] Shi, J., He, J., Zhu, H., Fang, H., Huang, Y., and Zhang, X. (2012). ORIENTAIS: Formal verified OSEK/VDX real-time operating system. In *Engineering of Complex Computer Systems*, pages 293–301. IEEE Computer Society.

[Sullivan and Masson, 1990] Sullivan, G. F. and Masson, G. M. (1990). Using certification trails to achieve software fault tolerance. In *FTCS*, pages 423–431. IEEE Computer Society.

[Thiemann and Sternagel, 2009] Thiemann, R. and Sternagel, C. (2009). Certification of termination proofs using CeTA. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 452–468.

[Verisoft XT, 2010] Verisoft XT (2010). Verisoft XT. http://www.verisoftxt.de.

[Wadler, 1990] Wadler, P. (1990). Linear types can change the world! In *Programming Concepts and Methods*. North.

[Wildmoser and Nipkow, 2004] Wildmoser, M. and Nipkow, T. (2004). Certifying machine code safety: Shallow versus deep embedding. In Slind, K., Bunker, A., and Gopalakrishnan, G., editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer.

[Winwood et al., 2009] Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., and Norrish, M. (2009). Mind the gap: A verification framework for low-level C. In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *TPHOLs*, volume 5674 of *LNCS*, pages 500–515. Springer.

Appendices

A

Isabelle Theories for Chapter 2

A.1. Witness Properties

A.1.1. Connected Components

```
theory Connected-Components
imports ../Graph-Theory/Graph-Theory
begin

locale connected-components-locale =
  fin-digraph +
  fixes num :: 'a ⇒ nat
  fixes parent-edge :: 'a ⇒ 'b option
  fixes r :: 'a
  assumes r-assms: r ∈ verts G ∧ parent-edge r = None ∧ num r = 0
  assumes parent-num-assms:
    ⋀v. v ∈ verts G ∧ v ≠ r ⟹
      ∃ e ∈ arcs G.
        parent-edge v = Some e ∧
        head G e = v ∧
        num v =  num (tail G e) + 1

sublocale connected-components-locale ⊆ fin-digraph G
  by auto

context connected-components-locale
begin
```

```
lemma ccl-wellformed: wf-digraph G
  by unfold-locales

lemma num-r-is-min:
  assumes v ∈ verts G
  assumes v ≠ r
  shows num v > 0
  using parent-num-assms assms
  by fastforce

lemma path-from-root:
  fixes v :: 'a
  assumes v ∈ verts G
  shows r →* v
  using assms
proof (induct num v arbitrary: v)
  case 0
  hence v = r using num-r-is-min by fastforce
  with ‹v ∈ verts G› show ?case by auto
next
  case (Suc n')
  hence v ≠ r using r-assms by auto
  then obtain e where ee:
    e ∈ arcs G
    head G e = v ∧ num v = num (tail G e) + 1
    using Suc parent-num-assms by blast
  with ‹v ∈ verts G› Suc(1,2) tail-in-verts
  have r →* (tail G e) tail G e → v
    by (auto intro: in-arcs-imp-in-arcs-ends)
  then show ?case by (rule reachable-adj-trans)
qed
```

The underlying undirected, simple graph is connected

```
lemma connectedG: connected G
proof (unfold connected-def, intro strongly-connectedI)
    show verts (with-proj (mk-symmetric G)) ≠ {}
          by (metis equals0D r-assms reachable-in-vertsE reachable-mk-symmetricI
reachable-refl)
  next
  let ?SG = mk-symmetric G
  interpret S: pair-fin-digraph ?SG ..
  fix u v assume uv-sG: u ∈ verts ?SG v ∈ verts ?SG
  from uv-sG have u ∈ verts G v ∈ verts G by auto
  then have u →*?SG r r →*?SG v
    by (auto intro: reachable-mk-symmetricI path-from-root  symmetric-reachable
```

```
      symmetric-mk-symmetric simp del: pverts-mk-symmetric)
  then show u →* ?SG v
    by (rule S.reachable-trans)
qed

theorem connected-by-path:
  fixes u v :: 'a
  assumes u ∈ pverts (mk-symmetric G)
  assumes v ∈ pverts (mk-symmetric G)
  shows u →* mk-symmetric G v
using connectedG  wellformed-mk-symmetric assms
unfolding connected-def strongly-connected-def by fastforce
end

corollary (in connected-components-locale) connected-graph:
  assumes u ∈ verts G and v ∈ verts G
  shows ∃p. vpath p (mk-symmetric G) ∧ hd p = u ∧ last p = v
proof –
  interpret S: pair-fin-digraph mk-symmetric G ..
  show ?thesis unfolding S.reachable-vpath-conv[symmetric]
    using assms by (auto intro: connected-by-path)
qed

end
```

A.1.2. Shortest Path

```
theory Shortest-Path-Theory
imports
  Complex
  ../Graph-Theory/Graph-Theory
begin


locale basic-sp =
  fin-digraph +
  fixes dist :: 'a ⇒ ereal
  fixes c :: 'b ⇒ real
  fixes s :: 'a
  assumes general-source-val: dist s ≤ 0
  assumes trian:
    ⋀e. e ∈ arcs G ⟹
      dist (head G e) ≤ dist (tail G e) + c e

locale basic-just-sp =
  basic-sp +
```

```
  fixes enum :: 'a ⇒ enat
  assumes just:
    ⋀v. ⟦v ∈ verts G; v ≠ s; enum v ≠ ∞⟧ ⟹
      ∃ e ∈ arcs G. v = head G e ∧
        dist v = dist (tail G e) + c e  ∧
        enum v = enum (tail G e) + (enat 1)

locale shortest-path-non-neg-cost =
  basic-just-sp +
  assumes s-in-G: s ∈ verts G
  assumes source-val: dist s = 0
  assumes no-path: ⋀v. v ∈ verts G ⟹ dist v = ∞ ⟷ enum v = ∞
  assumes non-neg-cost: ⋀e. e ∈ arcs G ⟹ 0 ≤ c e

locale basic-just-sp-pred =
  basic-sp +
  fixes enum :: 'a ⇒ enat
  fixes pred :: 'a ⇒ 'b option
  assumes just:
    ⋀v. ⟦v ∈ verts G; v ≠ s; enum v ≠ ∞⟧ ⟹
      ∃ e ∈ arcs G.
        e = the (pred v) ∧
        v = head G e ∧
        dist v = dist (tail G e) + c e  ∧
        enum v = enum (tail G e) + (enat 1)

sublocale basic-just-sp-pred ⊆ basic-just-sp
using basic-just-sp-pred-axioms
unfolding basic-just-sp-pred-def
   basic-just-sp-pred-axioms-def
by unfold-locales (blast)

locale shortest-path-non-neg-cost-pred =
  basic-just-sp-pred +
  assumes s-in-G: s ∈ verts G
  assumes source-val: dist s = 0
  assumes no-path: ⋀v. v ∈ verts G ⟹ dist v = ∞ ⟷ enum v = ∞
  assumes non-neg-cost: ⋀e. e ∈ arcs G ⟹ 0 ≤ c e

sublocale shortest-path-non-neg-cost-pred ⊆ shortest-path-non-neg-cost
using shortest-path-non-neg-cost-pred-axioms
by unfold-locales
   (auto simp: shortest-path-non-neg-cost-pred-def
   shortest-path-non-neg-cost-pred-axioms-def)

lemma tail-value-helper:
```

```
  assumes hd p = last p
  assumes distinct p
  assumes p ≠ []
  shows p = [hd p]
by (metis assms distinct.simps(2) append-butlast-last-id hd-append
  append-self-conv2 distinct-butlast hd-in-set not-distinct-conv-prefix)

lemma (in basic-sp) dist-le-cost:
  fixes v :: 'a
  fixes p :: 'b list
  assumes awalk s p v
  shows dist v ≤ awalk-cost c p
  using assms
  proof (induct length p arbitrary: p v)
  case 0
    hence s = v by auto
    thus ?case using 0(1) general-source-val
      by (metis awalk-cost-Nil length-0-conv zero-ereal-def)
  next
  case (Suc n)
    then obtain p' e where p'e: p = p' @ [e]
      by (cases p rule: rev-cases) auto
    then obtain u where ewu: awalk s p' u ∧ awalk u [e] v
      using awalk-append-iff Suc(3) by simp
    then have du: dist u ≤ ereal (awalk-cost c p')
      using Suc p'e by simp
    from ewu have ust: u = tail G e and vta: v = head G e
      by auto
    then have dist v ≤ dist u + c e
      using ewu du ust trian[where e=e] by force
    with du have dist v ≤ ereal (awalk-cost c p') + c e
      by (metis add-right-mono order-trans)
    thus dist v ≤ awalk-cost c p
      using awalk-cost-append p'e by simp
  qed

lemma (in fin-digraph) witness-path:
  assumes μ c s v = ereal r
  shows ∃ p. apath s p v ∧ μ c s v = awalk-cost c p
proof –
  have sv: s →* v
    using shortest-path-inf[of s v c] assms by fastforce
  {
    fix p assume awalk s p v
    then have no-neg-cyc:
    ¬ (∃ w q. awalk w q w ∧ w ∈ set (awalk-verts s p) ∧ awalk-cost c q < 0)
```

```
      using neg-cycle-imp-inf-μ assms by force
  }
  thus ?thesis using no-neg-cyc-reach-imp-path[OF sv] by presburger
qed

lemma (in basic-sp) dist-le-μ:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v ≤ μ c s v
proof (rule ccontr)
  assume nt: ¬ ?thesis
  show False
  proof (cases μ c s v)
    show ⋀r. μ c s v = ereal r ⟹ False
    proof –
      fix r assume r-asm: μ c s v = ereal r
      hence sv: s →* v
        using shortest-path-inf[where u=s and v=v and f=c] by auto
      obtain p where
        awalk s p v
        μ c s v = awalk-cost c p
        using witness-path[OF r-asm] unfolding apath-def by force
      thus False using nt dist-le-cost by simp
    qed
  next
    show μ c s v = ∞ ⟹ False using nt by simp
  next
    show μ c s v = – ∞ ⟹ False
    proof –
      assume asm: μ c s v = – ∞
      let ?C = (λx. ereal (awalk-cost c x)) ‘ {p. awalk s p v}
      have ∃ x∈ ?C. x < dist v
        using Inf-ereal-iff [where y =dist vand X=?C and z= −∞]
        nt asm unfolding μ-def INF-def by simp
      then obtain p where
        awalk s p v
        awalk-cost c p < dist v
        by force
      thus False using dist-le-cost by force
    qed
  qed
qed

lemma (in basic-just-sp) dist-ge-μ:
  fixes v :: 'a
  assumes v ∈ verts G
```

```
  assumes enum v ≠ ∞
  assumes dist v ≠ −∞
  assumes μ c s s = ereal 0
  assumes dist s = 0
  assumes ⋀u. u∈verts G ⟹ u≠s ⟹ enum u ≠ enat 0
  shows dist v ≥ μ c s v
proof –
  obtain n where enat n = enum v using assms(2) by force
  thus ?thesis using assms
  proof(induct n arbitrary: v)
  case 0 thus ?case by (cases v=s, auto)
  next
  case (Suc n)
    thus ?case
    proof (cases v=s)
    case False
      obtain e where e-assms:
        e ∈ arcs G
        v = head G e
        dist v = dist (tail G e) + ereal (c e)
        enum v = enum (tail G e) + enat 1
        using just[OF Suc(3) False Suc(4)] by blast
      then have nsinf:enum (tail G e) ≠ ∞
        by (metis Suc(2) enat.simps(3) enat-1 plus-enat-simps(2))
      then have ns:enat n = enum (tail G e)
        using e-assms(4) Suc(2) by force
      have  ds: dist (tail G e) = μ c s (tail G e)
        using Suc(1)[OF ns tail-in-verts[OF e-assms(1)] nsinf]
        Suc(5–8) e-assms(3) dist-le-μ[OF tail-in-verts[OF e-assms(1)]]
        by simp
      have dmuc:dist v = μ c s (tail G e) + ereal (c e)
        using e-assms(3) ds  by auto
      thus ?thesis
      proof (cases dist v = ∞)
      case False
        have arc-to-ends G e = (tail G e, v)
          unfolding arc-to-ends-def
          by (simp add: e-assms(2))
        obtain r where  μr: μ c s (tail G e) = ereal r
           using e-assms(3) Suc(5) ds False
           by (cases μ c s (tail G e), auto)
        obtain p where
          awalk s p (tail G e) and
          μs: μ c s (tail G e) = ereal (awalk-cost c p)
          using witness-path[OF μr] unfolding apath-def
          by blast
```

```
        then have pe: awalk s (p @ [e]) v
          using e-assms(1,2) by (auto simp: awalk-simps awlast-of-awalk)
        hence muc:μ c s v ≤ μ c s (tail G e) + ereal (c e)
        using μs min-cost-le-walk-cost[OF pe] by simp
        thus  dist v ≥ μ c s v  using dmuc by simp
      qed simp
    qed (simp add: Suc(6,7))
  qed
qed

lemma (in shortest-path-non-neg-cost) tail-value-check:
  fixes u :: 'a
  assumes s ∈ verts G
  shows μ c s s = ereal 0
proof −
  have ∗: awalk s [] s using assms unfolding awalk-def by simp
  hence μ c s s ≤ ereal 0 using min-cost-le-walk-cost[OF ∗] by simp
  moreover
  have (⋀p. awalk s p s ⟹ ereal(awalk-cost c p) ≥ ereal 0)
   using non-neg-cost pos-cost-pos-awalk-cost by auto
  hence μ c s s ≥ ereal 0
    unfolding μ-def by (blast intro: INF-greatest)
  ultimately
  show ?thesis by simp
qed

lemma (in shortest-path-non-neg-cost) enum-not0:
  fixes v :: 'a
  assumes v ∈ verts G
  assumes v ≠ s

  shows enum v ≠ enat 0
proof (cases enum v ≠ ∞)
case True
  then obtain ku where enum v = ku + enat 1
    using assms just by blast
  thus ?thesis  by (induct ku) auto
qed fast

lemma (in shortest-path-non-neg-cost) dist-ne-ninf:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v ≠ −∞
proof (cases enum v = ∞)
case False
  obtain n where enat n = enum v
```

```
    using False by force
  thus ?thesis using assms False
  proof(induct n arbitrary: v)
  case 0 thus ?case
    using enum-not0 source-val by (cases v=s, auto)
  next
  case (Suc n)
    thus ?case
    proof (cases v=s)
    case True
      thus ?thesis using source-val by simp
    next
    case False
      obtain e where e-assms:
        e ∈ arcs G
        dist v = dist (tail G e) + ereal (c e)
        enum v = enum (tail G e) + enat 1
        using just[OF Suc(3) False Suc(4)] by blast
      then have nsinf:enum (tail G e) ≠ ∞
        by (metis Suc(2) enat.simps(3) enat-1 plus-enat-simps(2))
      then have ns:enat n = enum (tail G e)
        using e-assms(3) Suc(2) by force
      have dist (tail G e ) ≠ − ∞
        by (rule Suc(1) [OF ns tail-in-verts[OF e-assms(1)] nsinf])
      thus ?thesis using e-assms(2) by simp
    qed
  qed
next
case True
  thus ?thesis using no-path[OF assms] by simp
qed

theorem (in shortest-path-non-neg-cost) correct-shortest-path:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v = μ c s v
  using no-path[OF assms(1)] dist-le-μ[OF assms(1)]
    dist-ge-μ[OF assms(1) - dist-ne-ninf[OF assms(1)]
    tail-value-check[OF s-in-G] source-val enum-not0]
    by fastforce

corollary (in shortest-path-non-neg-cost-pred) correct-shortest-path-pred:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v = μ c s v
  using correct-shortest-path assms by simp
```

end

A.1.3. Shortest Path with Arbitrary Edge Costs

```
theory Shortest-Path-Arbitrary-Edge-Costs


imports
  ../Graph-Theory/Graph-Theory
  Shortest-Path-Theory
begin


locale shortest-paths-init =
  fixes G :: ('a, 'b) pre-digraph (structure)
  fixes s :: 'a
  fixes c :: 'b ⇒ real
  fixes num :: 'a ⇒ nat
  fixes parent-edge :: 'a ⇒ 'b option
  fixes dist :: 'a  ⇒ ereal
  assumes graphG: fin-digraph G

abbreviation (in shortest-paths-init) V_f :: 'a set where
  V_f ≡ {v. v ∈ verts G ∧ (∃ r. dist v = ereal r)}

abbreviation (in shortest-paths-init) V_p :: 'a set where
  V_p ≡ {v. v ∈ verts G ∧ dist v = ∞}

abbreviation (in shortest-paths-init) V_n :: 'a set where
  V_n ≡ {v. v ∈ verts G ∧ dist v = −∞}

locale shortest-paths-reachable =
  shortest-paths-init +
  assumes s-assms:
    s ∈ verts G

    num s = 0
  assumes  pna:
    ⋀v.  ⟦v ∈ verts G; v ≠ s; v ∉ V_p⟧ ⟹
    (∃ e ∈ arcs G. parent-edge v = Some e ∧
    head G e = v ∧ tail G e ∉ V_p ∧
    num v  = num (tail G e) + 1)


sublocale shortest-paths-reachable ⊆ fin-digraph G
```

```
  using graphG by auto

definition (in shortest-paths-reachable) enum :: 'a ⇒ enat where
  enum v = (if (dist v = ∞ ∨ dist v = − ∞) then ∞ else num v)

locale shortest-paths-basic =
  shortest-paths-reachable +
  basic-just-sp G dist c s enum +
  assumes source-val: (∃ v ∈ verts G. enum v ≠ ∞) ⟹ dist s = 0


function (in shortest-paths-reachable) pwalk :: 'a ⇒ 'b list
where
  pwalk v =
    (if (v = s ∨ dist v = ∞ ∨ v ∉ verts G)
      then []
      else pwalk (tail G (the (parent-edge v))) @ [the (parent-edge v)]
    )
by auto
termination (in shortest-paths-reachable)
  using pna
  by (relation measure num, auto, fastforce)


lemma (in shortest-paths-reachable) pwalk-simps:
  v = s ∨ dist v = ∞ ∨ v ∉ verts G ⟹ pwalk v = []
  v ≠ s ⟹ dist v ≠ ∞ ⟹ v ∈ verts G ⟹
    pwalk v = pwalk (tail G (the (parent-edge v))) @ [the (parent-edge v)]
by auto

definition (in shortest-paths-reachable) pwalk-verts :: 'a  ⇒ 'a set where
  pwalk-verts v = {u. u ∈ set (awalk-verts s (pwalk v))}

locale shortest-paths-neg-cyc =
  shortest-paths-basic +
  fixes C :: ('a ×('b awalk)) set
  assumes C-se:
    C ⊆ {(u, p). dist u ≠ ∞ ∧ awalk u p u ∧ awalk-cost c p < 0}
  assumes int-neg-cyc:
    ⋀v. v ∈ V_n ⟹
      (fst ‘ C) ∩ pwalk-verts v  ≠ {}

locale shortest-paths-basic-pred =
  shortest-paths-reachable +
  fixes pred :: 'a ⇒ 'b option
  assumes bj: basic-just-sp-pred G dist c s enum pred
```

```
  assumes source-val: (∃ v ∈ verts G. enum v ≠ ∞) ⟹ dist s = 0


sublocale shortest-paths-basic-pred ⊆ shortest-paths-basic
  using shortest-paths-basic-pred-axioms
  unfolding shortest-paths-basic-pred-def shortest-paths-basic-pred-axioms-def
  shortest-paths-basic-def shortest-paths-basic-axioms-def
  basic-just-sp-pred-def basic-just-sp-pred-axioms-def
  basic-just-sp-def basic-just-sp-axioms-def
  by blast

lemma (in shortest-paths-reachable) num-s-is-min:
  assumes v ∈ verts G
  assumes v ≠ s
  assumes v ∉ V_p
  shows num v > 0
     using pna[OF assms] by fastforce


theorem (in shortest-paths-reachable) path-from-root-Vr-ex:
  fixes v :: 'a
  assumes v ∈ verts G
  assumes v ≠ s
  assumes v ∉ V_p
  shows  ∃ e. s →* tail G e ∧
          e ∈ arcs G ∧ head G e = v ∧ dist (tail G e) ≠ ∞ ∧
          parent-edge v = Some e ∧ num v = num (tail G e) + 1
using assms
proof(induct num v − 1 arbitrary : v)
case 0
  obtain e where ee:
    e ∈ arcs G
    head G e = v
    (tail G e) ∉ V_p
    parent-edge v = Some e
    num v = num (tail G e) + 1
    using pna[OF 0(2–4)]  by fast
  have tail G e = s
    using num-s-is-min[OF tail-in-verts [OF ee(1)] - ee(3)]
    ee(5) 0(1) by auto
  then show ?case using ee by auto
next
case (Suc n')
  obtain e where ee:
    e ∈ arcs G
```

$head\ G\ e = v$
$(tail\ G\ e) \notin V_p$
$parent\text{-}edge\ v = Some\ e$
$num\ v = num\ (tail\ G\ e) + 1$
using *pna*[*OF Suc*(*3–5*)] **by** *fast*
then have *ss*: $tail\ G\ e \neq s$
using *num-s-is-min tail-in-verts ee*
Suc(*2*) *s-assms*(*2*) **by** *force*
have *nst*: $n' = num\ (tail\ G\ e) - 1$
using *ee*(*5*) *Suc*(*2*) **by** *presburger*
obtain e' **where**
reach: $s \to^* tail\ G\ e'$ **and**
e': $e' \in arcs\ G \wedge head\ G\ e' = tail\ G\ e \wedge (tail\ G\ e') \notin V_p$
using *Suc*(*1*)[*OF nst tail-in-verts*[*OF ee*(*1*)] *ss ee*(*3*)] **by** *blast*
from *reach* **also have** $tail\ G\ e' \to tail\ G\ e$ **using** e'
by (*metis in-arcs-imp-in-arcs-ends*)
finally show *?case* **using** e' *ee* **by** *auto*
qed

corollary (**in** *shortest-paths-reachable*) *path-from-root-Vr*:
fixes $v :: 'a$
assumes $v \in verts\ G$
assumes $v \notin V_p$
shows $s \to^* v$
proof(*cases* $v = s$)
case *True* **thus** *?thesis* **using** *assms* **by** *simp*
next
case *False*
obtain e **where** $s \to^* tail\ G\ e$ **and** $e \in arcs\ G$ **and** $head\ G\ e = v$
using *path-from-root-Vr-ex*[*OF assms*(*1*) *False assms*(*2*)] **by** *blast*
then have $s \to^* tail\ G\ e$ **and** $tail\ G\ e \to v$
by (*auto intro*: *in-arcs-imp-in-arcs-ends*)
then show *?thesis* **by** (*rule reachable-adj-trans*)
qed

corollary (**in** *shortest-paths-reachable*) *not-Vp-μ-less-inf*:
fixes $v :: 'a$
assumes $v \in verts\ G$
assumes $v \notin V_p$
shows $\mu\ c\ s\ v \neq \infty$
using *assms path-from-root-Vr μ-reach-conv* **by** *force*

lemma (**in** *shortest-paths-basic*) *enum-not0*:
assumes $v \in verts\ G$

```
  assumes v ≠ s
  shows enum v ≠ enat 0
  using pna[OF assms(1,2)] assms unfolding enum-def by auto


lemma (in shortest-paths-basic) dist-Vf-μ:
  fixes v :: 'a
  assumes vG: v ∈ verts G
  assumes ∃ r. dist v = ereal r
  shows dist v = μ c s v
proof –
  have ds: dist s =  0
    using assms source-val unfolding enum-def by force
  have ews:awalk s [] s
    using s-assms(1) unfolding awalk-def by simp
  have mu: μ c s s = ereal 0
    using min-cost-le-walk-cost[OF ews, where c=c]
    awalk-cost-Nil ds  dist-le-μ[OF s-assms(1)] zero-ereal-def
    by simp
  thus ?thesis
    using ds assms dist-le-μ[OF vG]
    dist-ge-μ[OF vG - - mu ds enum-not0]
    unfolding enum-def by fastforce
qed

lemma (in shortest-paths-reachable) pwalk-awalk:
  fixes v :: 'a
  assumes v ∈ verts G
  assumes dist v ≠ ∞
  shows awalk s (pwalk v) v
proof (cases v=s)
case True
  thus ?thesis
    using assms pwalk.simps[where v=v]
    awalk-Nil-iff by presburger
next
case False
  from assms show ?thesis
  proof (induct rule: pwalk.induct)
    fix v
    let ?e = the (parent-edge v)
    let ?u = tail G ?e
    assume ewu: ¬ (v = s ∨ dist v = ∞ ∨ v ∉ verts G) ⟹
                ?u ∈ verts G ⟹ dist ?u ≠ ∞ ⟹
                awalk s (pwalk ?u) ?u
    assume vG: v ∈ verts G
```

```
    assume dv: dist v ≠ ∞
    thus awalk s (pwalk v) v
    proof (cases v = s ∨ dist v = ∞ ∨ v ∉ verts G)
    case True
      thus ?thesis
        using pwalk.simps vG dv
        awalk-Nil-iff by fastforce
    next
    case False
      obtain e  where ee:
        e ∈arcs G
        parent-edge v = Some e
        head G e = v
        (tail G e) ∉ V_p
        using pna False by blast
      hence awalk s (pwalk ?u) ?u
        using ewu[OF False] tail-in-verts by simp
      hence awalk s (pwalk (tail G e) @ [e]) v
        using ee(1−3) vG
        by (auto simp: awalk-simps simp del: pwalk.simps)
      thus ?thesis
        by (simp only: pwalk.simps[where v=v, unfolded ee(2), simplified False if-False option.sel])
    qed
  qed
qed

lemma (in shortest-paths-neg-cyc) Vn-μ-ninf:
  fixes v :: 'a
  assumes v ∈ V_n
  shows μ c s v = − ∞
proof −
  have awalk s (pwalk v) v
    using pwalk-awalk assms by force
moreover
  obtain w where ww: w ∈ fst ‘ C ∩ pwalk-verts v
    using int-neg-cyc[OF assms] by blast
  then obtain q where
    awalk w q w and
    awalk-cost c q < 0
    using C-se by auto
moreover
  have w ∈ set (awalk-verts s (pwalk v))
    using ww unfolding pwalk-verts-def by fast
ultimately
  show ?thesis using neg-cycle-imp-inf-μ by force
```

```
qed

theorem (in shortest-paths-neg-cyc) correct-shortest-path:
  fixes v :: 'a
  assumes v ∈ verts G
  shows dist v = μ c s v
proof(cases dist v)
show ⋀r. dist v = ereal r ⟹ dist v = μ c s v
  using dist-Vf-μ[OF assms] by simp
next
show dist v = ∞ ⟹ dist v = μ c s v
  using dist-le-μ[OF assms] by simp
next
show dist v = −∞ ⟹ dist v = μ c s v
  using Vn-μ-ninf assms by simp
qed

end
```

A.1.4. Maximum Cardinality Matching

```
theory Matching
imports
  Main
  Parity
  ../Graph-Theory/Graph-Theory
begin


type-synonym label = nat

definition disjoint-arcs :: ('a, 'b) pre-digraph => 'b ⇒ 'b ⇒ bool where
  disjoint-arcs G e1 e2 = (
    tail G e1 ≠ tail G e2 ∧ tail G e1 ≠ head G e2 ∧
    head G e1 ≠ tail G e2 ∧ head G e1 ≠ head G e2)

definition matching :: ('a, 'b) pre-digraph ⇒ 'b set ⇒ bool where
  matching G M = (M ⊆ arcs G ∧ (∀ e1 ∈ M. ∀ e2 ∈ M. e1 ≠ e2 ⟶ disjoint-arcs G
e1 e2))

definition OSC :: ('a, 'b) pre-digraph ⇒ ('a ⇒ label) ⇒ bool where
  OSC G L = (
    ∀ e ∈ arcs G.
      L (tail G e) = 1 ∨ L (head G e) = 1 ∨
      L (tail G e) = L (head G e) ∧ L (tail G e) ≥ 2)
```

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

definition $N :: {'a}\ \mathit{set} \Rightarrow ({'a} \Rightarrow \mathit{label}) \Rightarrow \mathit{label} \Rightarrow \mathit{nat}$ **where**
$N\ V\ L\ i \equiv \mathit{card}\ \{v \in V.\ L\ v = i\}$

locale $\mathit{matching\text{-}locale} = \mathit{digraph}\ +$
fixes $\mathit{maxM} :: {'b}\ \mathit{set}$
fixes $L :: {'a} \Rightarrow \mathit{label}$
assumes $\mathit{matching}$: $\mathit{matching}\ G\ \mathit{maxM}$
assumes OSC: $\mathit{OSC}\ G\ L$
assumes weight: $\mathit{card}\ \mathit{maxM} = \mathit{weight}\ \{i \in L\ `\ \mathit{verts}\ G.\ i > 1\}\ (N\ (\mathit{verts}\ G)\ L)$

sublocale $\mathit{matching\text{-}locale} \subseteq \mathit{digraph}$ **..**

context $\mathit{matching\text{-}locale}$ **begin**

definition $\mathit{degree} :: {'a} \Rightarrow \mathit{nat}$ **where**
$\mathit{degree}\ v \equiv \mathit{card}\ \{e \in \mathit{arcs}\ G.\ \mathit{tail}\ G\ e = v \vee \mathit{head}\ G\ e = v\}$

definition $\mathit{edge\text{-}as\text{-}set} :: {'b} \Rightarrow {'a}\ \mathit{set}$ **where**
$\mathit{edge\text{-}as\text{-}set}\ e \equiv \{\mathit{tail}\ G\ e,\ \mathit{head}\ G\ e\}$

definition $\mathit{matched} :: {'b}\ \mathit{set} \Rightarrow {'a} \Rightarrow \mathit{bool}$ **where**
$\mathit{matched}\ M\ v \equiv v \in \bigcup\ (\mathit{edge\text{-}as\text{-}set}\ `\ M)$

definition $\mathit{free} :: {'b}\ \mathit{set} \Rightarrow {'a} \Rightarrow \mathit{bool}$ **where**
$\mathit{free}\ M\ v \equiv \neg\ \mathit{matched}\ M\ v$

definition $\mathit{matching\text{-}i} :: \mathit{nat} \Rightarrow {'b}\ \mathit{set} \Rightarrow {'b}\ \mathit{set}$ **where**
$\mathit{matching\text{-}i}\ i\ M \equiv \{e \in M.\ i{=}1 \wedge (L\ (\mathit{tail}\ G\ e) = i \vee L\ (\mathit{head}\ G\ e) = i)$
$\vee\ i{>}1 \wedge L\ (\mathit{tail}\ G\ e) = i \wedge L\ (\mathit{head}\ G\ e) = i\}$

definition $\mathit{V\text{-}i}{::}\ \mathit{nat} \Rightarrow {'b}\ \mathit{set} \Rightarrow {'a}\ \mathit{set}$ **where**
$\mathit{V\text{-}i}\ i\ M \equiv \bigcup\ (\mathit{edge\text{-}as\text{-}set}\ `\ \mathit{matching\text{-}i}\ i\ M)$

definition $\mathit{endpoint\text{-}inV} :: {'a}\ \mathit{set} \Rightarrow {'b} \Rightarrow {'a}$ **where**
$\mathit{endpoint\text{-}inV}\ V\ e \equiv\ \mathit{if}\ \mathit{tail}\ G\ e \in V\ \mathit{then}\ \mathit{tail}\ G\ e\ \mathit{else}\ \mathit{head}\ G\ e$

definition $\mathit{relevant\text{-}endpoint} :: {'b} \Rightarrow {'a}$ **where**
$\mathit{relevant\text{-}endpoint}\ e \equiv \mathit{if}\ L\ (\mathit{tail}\ G\ e) = 1\ \mathit{then}\ \mathit{tail}\ G\ e\ \mathit{else}\ \mathit{head}\ G\ e$

lemma $\mathit{definition\text{-}of\text{-}range}$:
$\mathit{endpoint\text{-}inV}\ \mathit{V1}\ `\ \mathit{matching\text{-}i}\ 1\ M =$

```
  { v. ∃ e ∈ matching-i 1 M. endpoint-inV V1 e = v } by auto

lemma matching-i-arcs-as-sets:
  edge-as-set ‘ matching-i i M =
  { e1. ∃ e ∈ matching-i i M. edge-as-set e = e1} by auto

lemma matching-disjointness:
  assumes matching G M
  assumes e1 ∈ M
  assumes e2 ∈ M
  assumes e1 ≠ e2
  shows  edge-as-set e1 ∩ edge-as-set e2 = {}
  using assms
  by (auto simp add: edge-as-set-def disjoint-arcs-def matching-def)

lemma expand-set-containment:
  assumes matching G M
  assumes e ∈ M
  shows e ∈ arcs G
  using assms
  by (auto simp add:matching-def)

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

lemma card-M1-le-NVL1:
  assumes matching G M
  shows card (matching-i 1 M) ≤ N (verts G) L 1
proof –
  let ?f = endpoint-inV {v ∈ verts G. L v = 1}
  let ?A = matching-i 1 M
  let ?B = {v ∈ verts G. L v = 1}
  have inj-on ?f ?A using assms injectivity
```

```
    unfolding inj-on-def by blast
  moreover have ?f ‘ ?A ⊆ ?B
  proof –
    {
      fix e assume e ∈ matching-i 1 M
      hence e ∈ arcs G
        using assms by (auto simp add: matching-def matching-i-def)
      with ‹e ∈ matching-i 1 M›
      have endpoint-inV {v ∈ verts G. L v = 1} e ∈ {v ∈ verts G. L v = 1}
        using assms
            by (auto simp add: endpoint-inV-def matching-i-def intro: tail-in-verts head-in-verts)
    }
    then show ?thesis using assms definition-of-range by blast
  qed
  moreover have finite ?B by simp
  ultimately show ?thesis unfolding N-def by (rule card-inj-on-le)
qed

lemma edge-as-set-inj-on-Mi:
  assumes matching G M
  shows inj-on edge-as-set (matching-i i M)
  using assms
  unfolding inj-on-def edge-as-set-def matching-def
    disjoint-arcs-def matching-i-def
  by blast

lemma card-edge-as-set-Mi-twice-card-partitions:
  assumes matching G M ∧ i > 1
  shows 2 ∗ card (edge-as-set‘matching-i i M)
  = card (V-i i M) (is 2 ∗ card ?C = card ?Vi)
proof –
  from assms have 1: finite (⋃ ?C)
    by (auto simp add: matching-def
      matching-i-def edge-as-set-def finite-subset)
  show ?thesis unfolding V-i-def
  proof (rule card-partition)
    show finite ?C using 1 by (rule finite-UnionD)
  next
    show finite (⋃ ?C) using 1 .
  next
    fix c assume c ∈ ?C then show card c = 2
    proof (rule imageE)
      fix x
      assume 2: c = edge-as-set x and 3: x ∈ matching-i i M
      with assms have x ∈ arcs G
```

```
        unfolding matching-i-def matching-def by blast
      then have tail G x ≠ head G x using assms 3 by (metis no-loops)
      with 2 show ?thesis by (auto simp add: edge-as-set-def)
    qed
  next
    fix x1 x2
    assume 4: x1 ∈ ?C and 5: x2 ∈ ?C and 6: x1 ≠ x2
    {
      fix e1 e2
      assume 7: x1 = edge-as-set e1 e1 ∈ matching-i i M
        x2 = edge-as-set e2 e2 ∈ matching-i i M
      from assms have matching G M by simp
      moreover
      from 7 assms have e1 ∈ M and e2 ∈ M
        by (simp-all add: matching-i-def)
      moreover from 6 7 have e1 ≠ e2 by blast
      ultimately have x1 ∩ x2 = {} unfolding 7
        by (rule matching-disjointness)
    }
    with 4 5 show x1 ∩ x2 = {} by clarsimp
  qed
qed

lemma card-Mi-twice-card-Vi:
  assumes matching G M ∧ i > 1
  shows 2 * card (matching-i i M) = card (V-i i M)
proof –
  show ?thesis
    by (metis assms card-edge-as-set-Mi-twice-card-partitions
      edge-as-set-inj-on-Mi card-image)
qed

lemma card-Mi-le-floor-div-2-Vi:
  assumes matching G M ∧ i > 1
  shows card (matching-i i M) ≤ (card (V-i i M)) div 2
  using card-Mi-twice-card-Vi[OF assms]
  by arith

lemma card-Vi-le-NVLi:
  assumes i>1 ∧ matching G M
  shows card (V-i i M) ≤ N (verts G) L i
  unfolding N-def
proof (rule card-mono)
  show finite {v ∈ verts G. L v = i} using assms
    by (simp add: matching-def)
next
```

```
  let ?A = edge-as-set ‘ matching-i i M
  let ?C = {v ∈ verts G. L v = i}
  show V-i i M ⊆ ?C using assms unfolding V-i-def
  proof (intro Union-least)
    fix X assume X ∈ ?A
    with assms have ∃ x ∈ matching-i i M. edge-as-set x = X
      by (simp add: matching-i-arcs-as-sets)
    with assms show X ⊆ ?C
      unfolding matching-def
        matching-i-def edge-as-set-def by (blast intro: tail-in-verts head-in-verts)
  qed
qed

lemma card-Mi-le-floor-div-2-NVLi:
  assumes matching G M ∧ i > 1
  shows card (matching-i i M) ≤ (N (verts G) L i) div 2
proof –
  from assms have card (V-i i M) ≤ (N (verts G) L i)
    by (simp add: card-Vi-le-NVLi)
  then have card (V-i i M) div 2 ≤ (N (verts G) L i) div 2
    by simp
  moreover from assms have
    card (matching-i i M) ≤ card (V-i i M) div 2
    by (intro card-Mi-le-floor-div-2-Vi)
  ultimately show ?thesis by auto
qed

lemma card-M-le-sum-card-Mi:
assumes matching G M and OSC G L
shows card M ≤ (∑ i ∈ L‘verts G. card (matching-i i M))
  (is card - ≤ ?CardMi)
proof –
  let ?UnMi = ⋃ x ∈ L‘verts G. matching-i x M
  from assms have 1: finite ?UnMi
    by (auto simp add: matching-def matching-i-def finite-subset)
  {
    fix e assume e-inM: e ∈ M
    let ?v = relevant-endpoint e
    have 1: e ∈ matching-i (L ?v) M using assms e-inM
      proof cases
        assume L (tail G e) = 1
        thus ?thesis using assms e-inM
          by (simp add: relevant-endpoint-def matching-i-def)
      next
        assume a: L (tail G e) ≠ 1
        have L (tail G e) = 1 ∨ L (head G e) = 1
```

```
     ∨  (L (tail G e) = L (head G e) ∧ L (tail G e) >1)
     using assms e-inM unfolding OSC-def
     by (auto intro: expand-set-containment)
    thus ?thesis using assms e-inM a
     by (auto simp add: relevant-endpoint-def matching-i-def)
   qed
   have 2: ?v ∈ verts G using assms e-inM
     by (auto simp add: matching-def relevant-endpoint-def intro: tail-in-verts head-in-verts)
   then have ∃ v ∈ verts G. e ∈ matching-i (L v) M using assms 1 2
    by (intro bexI)
  }
  with assms have M ⊆ ?UnMi by (auto)
  with assms and 1 have card M ≤ card ?UnMi by (intro card-mono)
  moreover from assms have card ?UnMi = ?CardMi
  proof (intro card-UN-disjoint)
   show finite (L‘verts G) by simp
  next
   show ∀ i∈L‘verts G. finite (matching-i i M) using assms
    using finite-arcs
    unfolding matching-def matching-i-def
    by (blast intro: finite-subset finite-arcs)
  next
   show ∀ i ∈ L‘verts G. ∀ j ∈ L‘verts G. i ≠ j ⟶
    matching-i i M ∩ matching-i j M = {} using assms
    by (auto simp add: matching-i-def)
  qed
 ultimately show ?thesis by simp
qed

theorem card-M-le-weight-NVLi:
 assumes matching G M and OSC G L
 shows card M ≤ weight {i ∈ L ‘ verts G. i > 1} (N (verts G) L) (is - ≤ ?W)
proof -
 let ?M01 = ∑ i| i ∈ L ‘ verts G ∧ (i=1 ∨ i=0). card (matching-i i M)
 let ?Mgr1 = ∑ i| i ∈ L ‘ verts G ∧ 1 < i. card (matching-i i M)
 let ?Mi = ∑ i∈L ‘ verts G. card (matching-i i M)
 have card M ≤ ?Mi using assms by (rule card-M-le-sum-card-Mi)
 moreover
 have ?Mi ≤ ?W
 proof -
  let ?A = {i ∈ L ‘ verts G. i = 1 ∨ i = 0}
  let ?B = {i ∈ L ‘ verts G. 1 < i}
  let ?g = λ i. card (matching-i i M)
  let ?set01 = { i. i : L ‘ verts G & (i = 1 | i = 0)}
  have a: L ‘ verts G = ?A ∪ ?B using assms by auto
```

```
    have b: setsum ?g (?A ∪ ?B) = setsum ?g ?A + setsum ?g ?B
      by (auto intro: setsum.union-disjoint)
    have 1: ?Mi = ?M01+ ?Mgr1 using assms a b by simp
    moreover
    have 0: card (matching-i 0 M) = 0 using assms
      by (simp add: matching-i-def)
      have 2: ?M01 ≤ N (verts G) L 1
      proof cases
        assume a: 1 ∈ L ‘ verts G
        have ?M01 = card (matching-i 1 M)
        proof cases
          assume b: 0 ∈ L ‘ verts G
          with a assms have  ?set01 = {0, 1} by blast
          thus ?thesis using assms 0 by simp
        next
          assume b: 0 ∉ L ‘ verts G
          with a have ?set01 = {1} by (auto simp del:One-nat-def)
          thus ?thesis by simp
        qed
        thus ?thesis using assms a
          by (simp del: One-nat-def, intro card-M1-le-NVL1)
      next
        assume a: 1 ∉ L ‘ verts G
        show ?thesis
        proof cases
          assume b: 0 ∈ L ‘ verts G
          with a assms have  ?set01 = {0} by (auto simp del:One-nat-def)
          thus ?thesis using assms 0 by auto
        next
          assume b: 0 ∉ L ‘ verts G
          with a have ?set01 = {} by (auto simp del:One-nat-def)
            then have ?M01 = (∑ i∈{}. card (matching-i i M)) by auto
            thus ?thesis by simp
          qed
        qed
      moreover
      have 3: ?Mgr1 ≤ (∑ i|i∈L ‘ verts G ∧ 1 < i. N (verts G) L i div 2)
        using assms
        by (intro setsum-mono card-Mi-le-floor-div-2-NVLi, simp)
    ultimately
    show ?thesis using 1 2 3 assms by (simp add: weight-def)
  qed
  ultimately show ?thesis by simp
qed

theorem maximum-cardinality-matching:
```

```
  matching G M′ ⟶ card M′ ≤ card maxM
  using card-M-le-weight-NVLi OSC matching weight
  by simp

end

end
```

A.2. Verification of Imperative Simpl code

A.2.1. Connected Components

Implementation

```
theory Check-Connected-Impl
imports
  Vcg
  ../Witness-Property/Connected-Components
begin

type-synonym IVertex = nat
type-synonym IEdge-Id = nat
type-synonym IEdge = IVertex × IVertex
type-synonym IPEdge = IVertex ⇒ IEdge-Id option
type-synonym INum = IVertex ⇒ nat
type-synonym IGraph = nat × nat × (IEdge-Id ⇒ IEdge)

abbreviation ivertex-cnt :: IGraph ⇒ nat
  where ivertex-cnt G ≡ fst G

abbreviation iedge-cnt :: IGraph ⇒ nat
  where iedge-cnt G ≡ fst (snd G)

abbreviation iedges :: IGraph ⇒ IEdge-Id ⇒ IEdge
  where iedges G ≡ snd (snd G)

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)
ML ⟪ Toplevel.theory ⟫
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) ⦄
```

```
definition parent-num-assms-inv :: IGraph ⇒ IVertex ⇒ IPEdge ⇒ INum ⇒ nat ⇒
bool where
  parent-num-assms-inv G r p n k ≡ ∀ i < k. i ≠ r ⟶ (case p i of
      None ⇒ False
    | Some x ⇒ x < iedge-cnt G ∧ snd (iedges G x) = i ∧ n i = n (fst (iedges G x)) +
1)
```

```
procedures parent-num-assms (G :: IGraph, r :: IVertex, parent-edge :: IPEdge,
    num :: INum | R :: bool)
  where
    vertex :: IVertex
    edge-id :: IEdge-Id
  in
    ANNO (G,r,p,n).
      ⦃ ´G = G ∧ ´r = r ∧ ´parent-edge = p ∧ ´num = n ⦄
      ´R :== True ;;
      ´vertex :== 0 ;;
      TRY
        WHILE ´vertex < ivertex-cnt ´G
        INV ⦃ ´R = parent-num-assms-inv ´G ´r ´parent-edge ´num ´vertex
          ∧ ´G = G ∧ ´r = r ∧ ´parent-edge = p ∧ ´num = n
          ∧ ´vertex ≤ ivertex-cnt ´G⦄
        VAR MEASURE (ivertex-cnt ´G − ´vertex)
        DO
          IF (´vertex ≠ ´r) THEN
            IF ´parent-edge ´vertex = None THEN
```

```
          ´R :== False ;;
          THROW
        FI ;;
        ´edge-id :== the (´parent-edge ´vertex) ;;
        IF ´edge-id ≥ iedge-cnt ´G
            ∨ snd (iedges ´G ´edge-id) ≠ ´vertex
            ∨ ´num ´vertex ≠ ´num (fst (iedges ´G ´edge-id)) + 1 THEN
          ´R :== False ;;
          THROW
        FI
      FI ;;
      ´vertex :== ´vertex + 1
    OD
  CATCH SKIP END
 {| ´G = G ∧ ´r = r ∧ ´parent-edge = p ∧ ´num = n
   ∧ ´R = parent-num-assms-inv ´G ´r ´parent-edge ´num (ivertex-cnt ´G)|}

procedures check-connected (G :: IGraph, r :: IVertex, parent-edge :: IPEdge,
    num :: INum | R :: bool)
  where
    R1 :: bool
    R2 :: bool
    R3 :: bool
  in
    ´R1 :== CALL is-wellformed(´G) ;;
    ´R2 :== ´r < ivertex-cnt ´G ∧ ´num ´r = 0 ∧ ´parent-edge ´r = None ;;
    ´R3 :== CALL parent-num-assms(´G, ´r, ´parent-edge, ´num) ;;
    ´R :== ´R1 ∧ ´R2 ∧ ´R3

end
```

Verification

```
theory Check-Connected-Verification
imports Vcg Check-Connected-Impl
begin

definition no-loops :: ('a, 'b) pre-digraph ⇒ bool where
  no-loops G ≡ ∀ e ∈ arcs G. tail G e ≠ head G e

definition abs-IGraph :: IGraph ⇒ (nat, nat) pre-digraph where
  abs-IGraph G ≡ (| verts = {0..<ivertex-cnt G}, arcs = {0..<iedge-cnt G},
    tail = fst o iedges G, head = snd o iedges G |)

lemma verts-absI[simp]: verts (abs-IGraph G) = {0..<ivertex-cnt G}
  and arcs-absI[simp]: arcs (abs-IGraph G) = {0..<iedge-cnt G}
```

```
  and tail-absI[simp]: tail (abs-IGraph G) e = fst (iedges G e)
  and head-absI[simp]: head (abs-IGraph G) e = snd (iedges G e)
  by (auto simp: abs-IGraph-def)

lemma is-wellformed-inv-step:
  is-wellformed-inv G (Suc i) ⟷ is-wellformed-inv G i
      ∧ fst (iedges G i) < ivertex-cnt G ∧ snd (iedges G i) < ivertex-cnt G
  by (auto simp add: is-wellformed-inv-def less-Suc-eq)

lemma (in is-wellformed-impl) is-wellformed-spec:
  ∀ G. Γ ⊢t ⦃´G = G⦄ ´R :== PROC is-wellformed(´G) ⦃´R = is-wellformed-inv G
(iedge-cnt G)⦄
  apply vcg
  apply (auto simp: is-wellformed-inv-step)
  apply (auto simp: is-wellformed-inv-def)
  done

lemma parent-num-assms-inv-step:
  parent-num-assms-inv G r p n (Suc i) ⟷ parent-num-assms-inv G r p n i
    ∧ (i ≠ r ⟶ (case p i of
      None ⇒ False
    | Some x ⇒ x < iedge-cnt G ∧ snd (iedges G x) = i ∧ n i = n (fst (iedges G x)) +
1))
  by (auto simp: parent-num-assms-inv-def less-Suc-eq)

lemma (in parent-num-assms-impl) parent-num-assms-spec:
  ∀ G r p n. Γ ⊢t ⦃ ´G = G ∧ ´r = r ∧ ´parent-edge = p ∧ ´num = n⦄
    ´R :== PROC parent-num-assms(´G, ´r, ´parent-edge, ´num)
    ⦃ ´R = parent-num-assms-inv G r p n (ivertex-cnt G)⦄
  apply vcg
  apply (simp-all add: parent-num-assms-inv-step)
  apply (auto simp: parent-num-assms-inv-def)
  done

lemma connected-components-locale-eq-invariants:
⋀G r p n.
  connected-components-locale (abs-IGraph G) n p r =
    (is-wellformed-inv G (iedge-cnt G) ∧
    r < ivertex-cnt G ∧ n r = 0 ∧ p r = None ∧
    parent-num-assms-inv G r p n (ivertex-cnt G))
proof -
  fix G r p n
  let ?aG = abs-IGraph G
  have is-wellformed-inv G (iedge-cnt G) = fin-digraph ?aG
    unfolding is-wellformed-inv-def fin-digraph-def fin-digraph-axioms-def
      wf-digraph-def
```

```
      by auto
moreover
  have (∀ v ∈ verts ?aG. v ≠ r ⟶
    (∃ e ∈ arcs ?aG. p v = Some e ∧
    head ?aG e = v ∧
    n v =  n (tail ?aG e) + 1))
    = parent-num-assms-inv G r p n (ivertex-cnt G)
  proof –
    { fix i assume (case p i of None ⇒ False
        | Some x ⇒ x < iedge-cnt G ∧ snd (iedges G x) = i ∧ n i = n (fst (iedges G x))
+ 1)
      then have ∃ x∈{0..<iedge-cnt G}. p i = Some x ∧ snd (iedges G x) = i ∧ n i =
n (fst (iedges G x)) + 1
      by (case-tac p i) auto }
    then show ?thesis
      by (auto simp: parent-num-assms-inv-def)
  qed
ultimately
show  ?thesis G r p n
  unfolding connected-components-locale-def
  connected-components-locale-axioms-def by auto
qed

theorem (in check-connected-impl) check-connected-eq-locale:
  ∀ G r p n. Γ ⊢t ⦃ ´G = G ∧ ´r = r ∧ ´parent-edge = p ∧ ´num = n ⦄
     ´R :== PROC check-connected (´G, ´r, ´parent-edge, ´num)
    ⦃ ´R = connected-components-locale (abs-IGraph G) n p r⦄
by vcg (auto simp: connected-components-locale-eq-invariants)

lemma connected-components-locale-imp-correct:
  assumes connected-components-locale (abs-IGraph G)n p r
  assumes u ∈ pverts (mk-symmetric (abs-IGraph G))
  assumes v ∈ pverts (mk-symmetric (abs-IGraph G))
  shows ∃ p. pre-digraph.apath (mk-symmetric (abs-IGraph G)) u p v
proof –
  interpret S: pair-wf-digraph mk-symmetric (abs-IGraph G)
    by (intro wf-digraph.wellformed-mk-symmetric
        connected-components-locale.ccl-wellformed[OF assms(1)])
  show ?thesis
    using connected-components-locale.connected-by-path[OF assms]
    by (simp only: S.reachable-apath)
qed

theorem (in check-connected-impl) check-connected-spec:
  ⋀G r p n. Γ ⊢t ⦃ ´G = G ∧ ´r = r ∧ ´parent-edge = p ∧ ´num = n ⦄
     ´R :== PROC check-connected(´G, ´r, ´parent-edge, ´num)
```

```
  ⦃ ´R ⟶
      (∀ u ∈ pverts (mk-symmetric (abs-IGraph G)).
        ∀ v ∈ pverts (mk-symmetric (abs-IGraph G)).
          ∃ p. pre-digraph.apath (mk-symmetric (abs-IGraph G)) u p v)⦄
using connected-components-locale-eq-invariants
      connected-components-locale-imp-correct
by vcg metis

end
```

A.2.2. Shortest Path

Implementation

```
theory Check-Shortest-Path-Impl
imports
  Vcg
  ../Witness-Property/Shortest-Path-Theory
~~/src/HOL/Statespace/StateSpaceLocale
begin

type-synonym IVertex = nat
type-synonym IEdge-Id = nat
type-synonym IEdge = IVertex × IVertex
type-synonym ICost = IEdge-Id ⇒ nat
type-synonym IDist = IVertex ⇒ ereal
type-synonym IPEdge = IVertex ⇒ IEdge-Id option
type-synonym INum = IVertex ⇒ enat
type-synonym IGraph = nat × nat × (IEdge-Id ⇒ IEdge)

abbreviation ivertex-cnt :: IGraph ⇒ nat
  where ivertex-cnt G ≡ fst G

abbreviation iedge-cnt :: IGraph ⇒ nat
  where iedge-cnt G ≡ fst (snd G)

abbreviation iarcs :: IGraph ⇒ IEdge-Id ⇒ IEdge
  where iarcs G ≡ snd (snd G)

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


definition trian-inv :: IGraph ⇒ IDist ⇒ ICost ⇒ nat ⇒ bool where
  trian-inv G d c m ≡
    ∀ i < m. d (snd (iarcs G i)) ≤ d (fst (iarcs G i)) + ereal (c i)

procedures trian (G :: IGraph, dist :: IDist, c :: ICost | R :: bool)
  where
    edge-id :: IEdge-Id
  in
    ANNO (G,dist,c).
      ⦃ ´G = G ∧ ´dist = dist ∧ ´c = c ⦄
      ´R :== True ;;
      ´edge-id :== 0 ;;
      TRY
        WHILE ´edge-id < iedge-cnt ´G
        INV ⦃ ´R = trian-inv ´G ´dist ´c ´edge-id
          ∧ ´G = G ∧ ´dist = dist ∧ ´c = c
          ∧ ´edge-id ≤ iedge-cnt ´G⦄
        VAR MEASURE (iedge-cnt ´G − ´edge-id)
        DO
          IF  ´dist (snd (iarcs ´G ´edge-id)) >
              ´dist (fst (iarcs ´G ´edge-id)) +
              ereal (´c ´edge-id) THEN
            ´R :== False ;;
            THROW
```

```
        FI ;;
        ´edge-id :== ´edge-id + 1
      OD
    CATCH SKIP END
    ⦃ ´G = G ∧ ´dist = dist ∧ ´c = c
    ∧ ´R = trian-inv ´G ´dist ´c (iedge-cnt ´G) ⦄
```

```
definition just-inv ::
  IGraph ⇒ IDist ⇒ ICost ⇒ IVertex ⇒ INum ⇒ IPEdge ⇒ nat ⇒ bool where
  just-inv G d c s n p k ≡
    ∀ v < k. v ≠ s ∧ n v ≠ ∞ ⟶
      (∃ e. e = the (p v) ∧ e < iedge-cnt G ∧
        v = snd (iarcs G e) ∧
        d v = d (fst (iarcs G e)) + ereal (c e) ∧
        n v = n (fst (iarcs G e)) + (enat 1))
```

```
procedures just (G :: IGraph, dist :: IDist, c :: ICost,
    s :: IVertex, enum :: INum, pred :: IPEdge | R :: bool)
  where
    v :: IVertex
    edge-id :: IEdge-Id
  in
    ANNO (G,dist, c, s ,enum, pred).
      ⦃ ´G = G ∧ ´dist = dist ∧ ´c = c ∧ ´s = s ∧ ´enum = enum ∧ ´pred = pred⦄
      ´R :== True ;;
      ´v :== 0 ;;
      TRY
        WHILE ´v < ivertex-cnt ´G
        INV ⦃ ´R = just-inv ´G ´dist ´c ´s ´enum ´pred ´v
          ∧ ´G = G ∧ ´c = c ∧ ´s = s ∧ ´dist = dist
          ∧ ´enum = enum ∧ ´pred = pred
          ∧ ´v ≤ ivertex-cnt ´G⦄
        VAR MEASURE (ivertex-cnt ´G − ´v)
        DO
          ´edge-id :== the (´pred ´v) ;;
          IF (´v ≠ ´s) ∧  ´enum ´v ≠ ∞ ∧
             (´edge-id ≥ iedge-cnt ´G
              ∨ snd (iarcs ´G ´edge-id) ≠ ´v
              ∨ ´dist ´v ≠
                 ´dist (fst (iarcs ´G ´edge-id)) + ereal (´c ´edge-id)
              ∨ ´enum ´v ≠ ´enum (fst (iarcs ´G ´edge-id)) + (enat 1)) THEN
            ´R :== False ;;
            THROW
          FI;;
          ´v :== ´v + 1
```

```
    OD
   CATCH SKIP END
 ⦃ ´G = G ∧ ´dist = dist ∧ ´c = c ∧ ´s = s ∧ ´enum = enum ∧ ´pred = pred
   ∧ ´R = just-inv ´G ´dist ´c ´s ´enum ´pred (ivertex-cnt ´G) ⦄


definition no-path-inv :: IGraph ⇒ IDist ⇒ INum ⇒ nat ⇒ bool where
  no-path-inv G d n k ≡ ∀ v < k. (d v = ∞ ⟷ n v = ∞)

procedures no-path (G :: IGraph, dist :: IDist, enum :: INum | R :: bool)
  where
    v :: IVertex
  in
    ANNO (G,dist,enum).
     ⦃ ´G = G ∧ ´dist = dist ∧ ´enum = enum ⦄
     ´R :== True ;;
     ´v :== 0 ;;
     TRY
       WHILE ´v < ivertex-cnt ´G
       INV ⦃ ´R = no-path-inv ´G ´dist ´enum ´v
         ∧ ´G = G ∧ ´dist = dist ∧ ´enum = enum
         ∧ ´v ≤ ivertex-cnt ´G⦄
       VAR MEASURE (ivertex-cnt ´G − ´v)
       DO
         IF ¬(´dist ´v = ∞ ⟷ ´enum ´v = ∞) THEN
           ´R :== False ;;
           THROW
         FI ;;
         ´v :== ´v + 1
       OD
     CATCH SKIP END
     ⦃ ´G = G ∧ ´dist = dist ∧ ´enum = enum
     ∧ ´R = no-path-inv ´G ´dist ´enum (ivertex-cnt ´G) ⦄


definition non-neg-cost-inv :: IGraph ⇒ ICost ⇒ nat ⇒ bool where
  non-neg-cost-inv G c m ≡ ∀ e < m. c e ≥ 0

procedures non-neg-cost (G :: IGraph, c :: ICost | R :: bool)
  where
    edge-id :: IEdge-Id
  in
    ANNO (G,c).
     ⦃ ´G = G ∧ ´c = c ⦄
     ´R :== True ;;
     ´edge-id :== 0 ;;
```

```
  TRY
    WHILE ´edge-id < iedge-cnt ´G
    INV ⦃ ´R = non-neg-cost-inv ´G ´c ´edge-id
      ∧ ´G = G ∧ ´c = c
      ∧ ´edge-id ≤ iedge-cnt ´G⦄
    VAR MEASURE (iedge-cnt ´G − ´edge-id)
    DO
      IF ´c ´edge-id < 0 THEN
        ´R :== False ;;
        THROW
      FI ;;
      ´edge-id :== ´edge-id + 1
    OD
  CATCH SKIP END
  ⦃ ´G = G ∧ ´c = c
  ∧ ´R = non-neg-cost-inv ´G ´c (iedge-cnt ´G) ⦄
```

```
procedures check-basic-just-sp (G :: IGraph, dist :: IDist, c :: ICost,
    s :: IVertex, enum :: INum, pred :: IPEdge | R :: bool)
  where
    R1 :: bool
    R2 :: bool
    R3 :: bool
    R4 :: bool
  in
    ´R1 :== CALL is-wellformed (´G) ;;
    ´R2 :== ´dist ´s ≤ 0 ;;
    ´R3 :== CALL trian (´G, ´dist, ´c) ;;
    ´R4 :== CALL just (´G, ´dist, ´c, ´s, ´enum, ´pred) ;;
    ´R :== ´R1 ∧ ´R2 ∧ ´R3 ∧ ´R4
```

```
procedures check-sp (G :: IGraph, dist :: IDist, c :: ICost,
    s :: IVertex, enum :: INum, pred :: IPEdge | R :: bool)
  where
    R1 :: bool
    R2 :: bool
    R3 :: bool
    R4 :: bool
  in
    ´R1 :== CALL check-basic-just-sp (´G, ´dist, ´c, ´s, ´enum, ´pred) ;;
    ´R2 :== ´s < ivertex-cnt ´G ∧ ´dist ´s = 0 ;;
    ´R3 :== CALL no-path (´G, ´dist, ´enum) ;;
    ´R4 :== CALL non-neg-cost (´G, ´c) ;;
    ´R :== ´R1 ∧ ´R2 ∧ ´R3 ∧ ´R4
```

```
end
```

Verification

```
theory Check-Shortest-Path-Verification
imports
  Vcg
  ../Simpl-Verification/Check-Shortest-Path-Impl

begin

definition no-loops :: ('a, 'b) pre-digraph ⇒ bool where
  no-loops G ≡ ∀ e ∈ arcs G. tail G e ≠ head G e

definition abs-IGraph :: IGraph ⇒ (nat, nat) pre-digraph where
  abs-IGraph G ≡ (| verts = {0..<ivertex-cnt G}, arcs = {0..<iedge-cnt G},
    tail = fst o iarcs G, head = snd o iarcs G |)

lemma verts-absI[simp]: verts (abs-IGraph G) = {0..<ivertex-cnt G}
  and arcs-absI[simp]: arcs (abs-IGraph G) = {0..<iedge-cnt G}
  and tail-absI[simp]: tail (abs-IGraph G) e = fst (iarcs G e)
  and head-absI[simp]: head (abs-IGraph G) e = snd (iarcs G e)
  by (auto simp: abs-IGraph-def)

lemma is-wellformed-inv-step:
  is-wellformed-inv G (Suc i) ⟷ is-wellformed-inv G i
      ∧ fst (iarcs G i) < ivertex-cnt G ∧ snd (iarcs G i) < ivertex-cnt G
  by (auto simp add: is-wellformed-inv-def less-Suc-eq)

lemma (in is-wellformed-impl) is-wellformed-spec:
  ∀ G. Γ ⊢_t {| ´G = G|} ´R :== PROC is-wellformed(´G) {|´R = is-wellformed-inv G
(iedge-cnt G)|}
  apply vcg
  apply (auto simp: is-wellformed-inv-step)
  apply (auto simp: is-wellformed-inv-def)
done

lemma trian-inv-step:
  trian-inv G d c (Suc i) ⟷ trian-inv G d c i
    ∧ d (snd (iarcs G i)) ≤ d (fst (iarcs G i)) + c i
  by (auto simp: trian-inv-def less-Suc-eq)

lemma (in trian-impl) trian-spec:
  ∀ G d c. Γ ⊢_t {| ´G = G ∧ ´dist = d ∧ ´c = c|}
```

```
    ´R :== PROC trian(´G, ´dist, ´c)
   ⦃ ´R = trian-inv G d c (iedge-cnt G)⦄
  apply vcg
  apply (auto simp add: trian-inv-step)
  apply (auto simp: trian-inv-def)
done

lemma just-inv-step:
  just-inv G d c s n p (Suc v) ⟷ just-inv G d c s n p v
    ∧ (v ≠ s ∧ n v ≠ ∞ ⟶
      (∃ e. e = the (p v) ∧ e < iedge-cnt G ∧
        v = snd (iarcs G e) ∧
        d v = d (fst (iarcs G e)) + ereal (c e) ∧
        n v = n (fst (iarcs G e)) + (enat 1)))
  by (auto simp: just-inv-def less-Suc-eq)

lemma just-inv-le:
  assumes j ≤ i just-inv G d c s n p i
  shows just-inv G d c s n p j
  using assms by (induct rule: dec-induct) (auto simp: just-inv-step)

lemma not-just-verts:
  fixes G R c d n p s v
  assumes v < ivertex-cnt G
  assumes v ≠ s ∧ n v ≠ ∞ ∧
        (iedge-cnt G ≤ the (p v) ∨
        snd (iarcs G (the (p v))) ≠ v ∨
        d v ≠
          d (fst (iarcs G (the (p v)))) + ereal (c (the (p v))) ∨
        n v ≠ n (fst (iarcs G (the (p v)))) + enat 1)
  shows ¬ just-inv G d c s n p (ivertex-cnt G)
proof (rule notI)
  assume jv: just-inv G d c s n p (ivertex-cnt G)
  have just-inv G d c s n p (Suc v)
    using just-inv-le[OF - jv] assms(1) by simp
  then have (v ≠ s ∧ n v ≠ ∞ ⟶
      (∃ e. e = the (p v) ∧ e < iedge-cnt G ∧
        v = snd (iarcs G e) ∧
        d v = d (fst (iarcs G e)) + ereal (c e) ∧
        n v = n (fst (iarcs G e)) + (enat 1)))
        by (auto simp: just-inv-step)
  with assms show False by force
qed

lemma (in just-impl) just-spec:
  ∀ G d c s n p.
```

```
  Γ ⊢t ⦃´G = G ∧ ´dist = d ∧
   ´c = c ∧ ´s = s ∧ ´enum = n ∧ ´pred = p⦄
   ´R :== PROC just(´G, ´dist, ´c, ´s, ´enum, ´pred)
  ⦃ ´R = just-inv  G d c s n p (ivertex-cnt G)⦄
 apply vcg
 apply (auto simp: not-just-verts just-inv-step)
 apply (simp add: just-inv-def)
done

lemma no-path-inv-step:
 no-path-inv G d n (Suc v) ⟷ no-path-inv G d n v
  ∧ (d v = ∞ ⟷ n v = ∞)
 by (auto simp add: no-path-inv-def less-Suc-eq)

lemma (in no-path-impl) no-path-spec:
 ∀ G d n. Γ ⊢t ⦃ ´G = G ∧ ´dist = d ∧ ´enum = n⦄
   ´R :== PROC no-path(´G, ´dist, ´enum)
  ⦃ ´R = no-path-inv G d n (ivertex-cnt G)⦄
 apply vcg
 apply (simp-all add: no-path-inv-step)
 apply (auto simp: no-path-inv-def)
done

lemma non-neg-cost-inv-step:
 non-neg-cost-inv G c (Suc i) ⟷ non-neg-cost-inv G c i
  ∧ c i ≥ 0
 by (auto simp add: non-neg-cost-inv-def less-Suc-eq)

lemma (in non-neg-cost-impl) non-neg-cost-spec:
 ∀ G c. Γ ⊢t ⦃ ´G = G ∧ ´c = c⦄
   ´R :== PROC non-neg-cost(´G, ´c)
  ⦃ ´R = non-neg-cost-inv G c (iedge-cnt G)⦄
 apply vcg
 apply (simp-all add: non-neg-cost-inv-step)
 apply (auto simp: non-neg-cost-inv-def)
done

lemma basic-just-sp-eq-invariants:
⋀G dist c s enum pred.
 basic-just-sp-pred (abs-IGraph G) dist c s enum pred ⟷
  (is-wellformed-inv G (iedge-cnt G) ∧
  dist s ≤ 0 ∧
  trian-inv G dist c (iedge-cnt G) ∧
  just-inv G dist c s enum pred (ivertex-cnt G))
proof −
 fix G d c s n p
```

```
  let ?aG = abs-IGraph G
  have fin-digraph (abs-IGraph G) ⟷ is-wellformed-inv G (iedge-cnt G)
    unfolding is-wellformed-inv-def fin-digraph-def fin-digraph-axioms-def
      wf-digraph-def no-loops-def
      by auto
moreover
  have trian-inv G d c (iedge-cnt G) =
    (∀ e. e ∈ arcs (abs-IGraph G) ⟶
   (d (head ?aG e) ≤ d (tail ?aG e) + ereal (c e)))
    by (simp add: trian-inv-def)
moreover
  have just-inv  G d c s n p (ivertex-cnt G) =
    (∀ v. v ∈ verts ?aG ⟶
      v ≠ s ⟶ n v ≠ ∞ ⟶
      (∃ e∈arcs ?aG. e = the (p v) ∧
      v = head ?aG e ∧
      d v = d (tail ?aG e) + ereal (c e) ∧
     n v = n (tail ?aG e) + enat 1))
      unfolding just-inv-def by fastforce
ultimately
   show ?thesis G d c s n p
   unfolding
    basic-just-sp-pred-def
    basic-just-sp-pred-axioms-def
    basic-sp-def basic-sp-axioms-def
   by presburger
qed

lemma (in check-basic-just-sp-impl) check-basic-just-sp-imp-locale:
  ∀  G d c s n p . Γ ⊢t ⦃ ´G = G ∧ ´dist = d ∧ ´c = c ∧ ´s = s ∧ ´enum = n ∧ ´pred
= p ⦄
     ´R :== PROC check-basic-just-sp (´G, ´dist, ´c, ´s, ´enum, ´pred)
    ⦃ ´R =  basic-just-sp-pred (abs-IGraph G) d c s n p⦄
    by vcg (simp add: basic-just-sp-eq-invariants)


lemma shortest-path-non-neg-cost-eq-invariants:
⋀G d c s n p .
  shortest-path-non-neg-cost-pred (abs-IGraph G) d c s n p ⟷
    (is-wellformed-inv G (iedge-cnt G) ∧
    d s ≤ 0 ∧
    trian-inv G d c (iedge-cnt G) ∧
    just-inv G d c s n p (ivertex-cnt G) ∧
    s < ivertex-cnt G ∧ d s = 0 ∧
    no-path-inv G d n (ivertex-cnt G) ∧
    non-neg-cost-inv G c (iedge-cnt G))
```

```
proof –
  fix G d c s n p
  let ?aG = abs-IGraph G
  have no-path-inv G d n (ivertex-cnt G) ⟷
    (∀ v. v ∈ verts ?aG ⟶ (d v = ∞) = (n v = ∞))
    by (simp add: no-path-inv-def)
moreover
  have non-neg-cost-inv G c (iedge-cnt G) ⟷
    (∀ e. e ∈ arcs ?aG ⟶ 0 ≤ c e)
    by (simp add: non-neg-cost-inv-def)
ultimately
   show ?thesis G d c s n p
   unfolding shortest-path-non-neg-cost-pred-def
    shortest-path-non-neg-cost-pred-axioms-def
   using basic-just-sp-eq-invariants by simp
qed

theorem (in check-sp-impl) check-sp-eq-locale:
 ∀ G d c s n p . Γ ⊢t ⦃ ´G = G ∧ ´dist = d ∧ ´c = c ∧ ´s = s ∧ ´enum = n ∧ ´pred
= p ⦄
     ´R :== PROC check-sp(´G, ´dist, ´c, ´s, ´enum, ´pred)
    ⦃ ´R = shortest-path-non-neg-cost-pred (abs-IGraph G) d c s n p⦄
    by vcg (auto simp add: shortest-path-non-neg-cost-eq-invariants)

lemma shortest-path-non-neg-cost-imp-correct:
⋀G d c s n p .
  shortest-path-non-neg-cost-pred (abs-IGraph G) d c s n p ⟶
   (∀ v ∈ verts (abs-IGraph G).
   d v = wf-digraph.μ (abs-IGraph G) c s v)
using shortest-path-non-neg-cost-pred.correct-shortest-path-pred by fast

theorem (in check-sp-impl) check-sp-spec:
 ∀ G d c s n p . Γ ⊢t ⦃ ´G = G ∧ ´dist = d ∧ ´c = c ∧ ´s = s ∧ ´enum = n ∧ ´pred
= p ⦄
     ´R :== PROC check-sp(´G, ´dist, ´c, ´s, ´enum, ´pred)
    ⦃ ´R ⟶ (∀ v ∈ verts (abs-IGraph G). d v = wf-digraph.μ (abs-IGraph G) c s v)⦄
using shortest-path-non-neg-cost-eq-invariants
      shortest-path-non-neg-cost-imp-correct
by vcg blast

end
```

A.3. Verification of C code within Isabelle/HOL

A.3.1. Connected Components

theory *Check-Connected*
imports
  ../Library/Autocorres-Misc
  ../Witness-Property/Connected-Components
begin

install-C-file *check-connected.c*

autocorres *check-connected.c*

context *check-connected* **begin**

lemma *validNFE-getsE*[*wp*]:
  ⦃λ*s. P* (*f s*) *s*⦄ *getsE f* ⦃*P*⦄, ⦃*E*⦄!
  by (*auto simp*: *getsE-def*) *wp*

lemma *validNFE-guardE*[*wp*]:
  ⦃λ*s. f s* $\wedge$ *P* () *s*⦄ *guardE f* ⦃*P*⦄, ⦃*Q*⦄!
  by (*auto simp*: *guardE-def*, *wp*, *linarith*)

lemma *eq-of-nat-conv*:
  assumes *unat w1* = *n*
  shows *w2* = *of-nat n* $\longleftrightarrow$ *w2* = *w1*
  using *assms* **by** *auto*

lemma *less-unat-plus1*:
  assumes *a* < *unat* (*b* + *1*)
  shows *a* < *unat b* $\vee$ *a* = *unat b*
  apply (*subgoal-tac b* + *1* $\neq$ *0*)
  using *assms unat-minus-one add-diff-cancel*
  by *fastforce*+

lemma *unat-minus-plus1-less*:
  fixes *a b*
  assumes *a* < *b*
  shows *unat* (*b* − (*a* + *1*)) < *unat* (*b* − *a*)

```
  by (metis (no-types) ab-semigroup-add-class.add-ac(1) right-minus-eq measure-unat
     add-diff-cancel2 assms is-num-normalize(1) zadd-diff-inverse linorder-neq-iff)

lemma unat-image-upto:
  fixes n :: 32 word
  shows unat ‘ {0..<n} = {unat 0..<unat n} (is ?A = ?B)
proof
  show ?B ⊆ ?A
  proof
    fix i assume a: i ∈ ?B
    then obtain i'∷ 32 word where ii: i=  unat i'
      by (metis ex-nat-less-eq le-unat-uoi not-leE order-less-asym unat-0)
    then have i' ∈ {0..<n}
      by (metis (hide-lams, mono-tags) atLeast0LessThan a unat-0
          word-zero-le lessThan-iff not-leE not-less-iff-gr-or-eq
          order-antisym  word-le-nat-alt Un-iff ivl-disj-un(8))
    thus  i ∈ ?A using ii by fast
  qed
next
  show ?A ⊆ ?B
  proof
     fix i assume a: i ∈ ?A
    then obtain i'∷ 32 word where ii: i=  unat i' by blast
    then have i' ∈ {0..<n} using a by force
    thus  i ∈ ?B
      by (metis Un-iff atLeast0LessThan ii ivl-disj-un(8)
          lessThan-iff unat-0 unat-mono word-zero-le)
  qed
qed


type-synonym IVertex = 32 word
type-synonym IEdge-Id = 32 word
type-synonym IEdge = IVertex × IVertex
type-synonym IPEdge = IVertex ⇒ 32 word
type-synonym INum = IVertex ⇒ 32 word
type-synonym IGraph = 32 word × 32 word × (IEdge-Id ⇒ IEdge)

abbreviation
  ivertex-cnt :: IGraph ⇒ 32 word
where
  ivertex-cnt G ≡ fst G

abbreviation
  iedge-cnt :: IGraph ⇒ 32 word
where
```

iedge-cnt G ≡ fst (snd G)

abbreviation
iedges :: IGraph ⇒ IEdge-Id ⇒ IEdge
where
iedges G ≡ snd (snd G)

fun
bool::32 word ⇒ bool
where
bool b = (if b=0 then False else True)

fun
mk-list′ :: nat ⇒ (32 word ⇒ ′b) ⇒ ′b list
where
mk-list′ n f = map f (map of-nat [0..<n])

fun
mk-list′-temp :: nat ⇒ (32 word ⇒ ′b) ⇒ nat ⇒ ′b list
where
mk-list′-temp 0 - - = [] |
mk-list′-temp (Suc x) f i = (f (of-nat i)) # mk-list′-temp x f (Suc i)

fun
mk-iedge-list :: IGraph ⇒ IEdge list
where
mk-iedge-list G = mk-list′ (unat (iedge-cnt G)) (iedges G)

fun
mk-inum-list :: IGraph ⇒ INum ⇒ 32 word list
where
mk-inum-list G num = mk-list′ (unat (ivertex-cnt G)) num

fun
mk-ipedge-list :: IGraph ⇒ IPEdge ⇒ 32 word list
where
mk-ipedge-list G pedge = mk-list′ (unat (ivertex-cnt G)) pedge

fun
to-edge :: IEdge ⇒ Edge-C
where
to-edge (u,v) = Edge-C u v

```
lemma s-C-pte[simp]:
  s-C (to-edge e) = fst e
  by (cases e) auto

lemma t-C-pte[simp]:
  t-C (to-edge e) = snd e
  by (cases e) auto

definition is-graph where
  is-graph h iG p ≡
    is-valid-Graph-C h p ∧
    ivertex-cnt iG = n-C (heap-Graph-C h p) ∧
    iedge-cnt iG = m-C (heap-Graph-C h p) ∧
    arrlist (heap-Edge-C h) (is-valid-Edge-C h)
      (map to-edge (mk-iedge-list iG)) (es-C (heap-Graph-C h p))

definition
  is-numm h iG iN p ≡ arrlist (heap-w32 h) (is-valid-w32 h) (mk-inum-list iG iN) p

definition
  is-pedge h iG iP (p:: 32 signed word ptr) ≡ arrlist (λp. heap-w32 h (ptr-coerce p))
        (λp. is-valid-w32 h (ptr-coerce p)) (mk-ipedge-list iG iP) p

lemma sint-ucast:
  sint (ucast (x ::word32) :: sword32) = sint x
  by (clarsimp simp: sint-uint uint-up-ucast is-up)

definition
  is-root :: IGraph ⇒ IVertex ⇒ IPEdge ⇒ INum ⇒ bool
where
  is-root iG r iP iN ≡ r < (ivertex-cnt iG) ∧ (iN r = 0) ∧ (sint (iP r) < 0)

definition
  parent-num-assms-inv :: IGraph ⇒ IVertex ⇒ IPEdge ⇒ INum ⇒ nat ⇒ bool
where
  parent-num-assms-inv G r p n k ≡
    ∀ i < k. (of-nat i) ≠ r ⟶
          0 ≤ sint (p (of-nat i)) ∧
          ((p (of-nat i)) < iedge-cnt G ∧
          snd (iedges G (p (of-nat i))) = (of-nat i) ∧
          n (of-nat i) = n (fst (iedges G (p (of-nat i)))) + 1) ∧
          n (of-nat i) < ivertex-cnt G

function (in connected-components-locale)
  pwalk :: 'a ⇒ 'a list
where
```

```
  pwalk v =
    (if (v = r  ∨ v ∉ verts G)
      then [v]
      else
       pwalk (tail G (the (parent-edge v))) ⊕ [tail G (the (parent-edge v)), v])
  by simp+
termination (in connected-components-locale)
  using parent-num-assms
  by (relation measure num, auto, fastforce)

lemma (in connected-components-locale) pwalk-simps:
  v = r ∨ v ∉ verts G ⟹ pwalk v = [v]
  v ≠ r ⟹ v ∈ verts G ⟹ pwalk v =
    pwalk (tail G (the (parent-edge v))) @ [v]
  by (simp, metis drop-0 pwalk.simps
    drop-Suc-Cons vwalk-join-def drop-Suc)

lemma (in connected-components-locale) pwalk-ne: pwalk v ≠ []
  by (metis drop-0 drop-Suc drop-Suc-Cons not-Cons-self
      pwalk.simps snoc-eq-iff-butlast vwalk-join-def)

lemma (in connected-components-locale) vwalk-length-pwalk:
  assumes v ∈ verts G
  assumes v ≠ r
  shows vwalk-length (pwalk v) =
         vwalk-length (pwalk (tail G (the (parent-edge v)))) + 1
  by (smt append-Cons assms length-append length-tl list.size(3,4) pwalk-ne
     pwalk.simps tl-append2 vwalk-join-Cons vwalk-join-def vwalk-length-simp)

lemma (in connected-components-locale) pwalk-split:
  assumes x ∈ set (pwalk v)
  shows ∃ p. pwalk v = pwalk x @ p
using assms
proof (induct vwalk-length (pwalk v) arbitrary: v)
case (Suc n)
have vnr: v ≠ r
  using Suc(2) by fastforce
show ?case
  proof (cases v ∈ verts G)
  case True
    thus ?thesis
    proof (cases x = v)
    case False
      let ?u = tail G (the (parent-edge v))
      have xpu: x ∈ set (pwalk ?u)
        using Suc(3) pwalk-simps(2)[OF vnr True] False by fastforce
```

```
      hence ∃p. pwalk (tail G (the (parent-edge v))) = pwalk x @ p
        using vwalk-length-pwalk[OF True vnr] Suc(2)
        by (metis Suc(1)[OF - xpu] Suc-eq-plus1
            Suc-eq-plus1-left diff-add-inverse)
      thus ?thesis using pwalk-simps(2)[OF vnr True] by fastforce
    qed fast
  qed (metis Suc.prems append-Nil2 empty-iff empty-set pwalk-simps(1) set-ConsD)
qed (metis pwalk-simps(1) add-is-0 vwalk-length-pwalk
     append-Nil2 empty-iff empty-set one-neq-zero set-ConsD)

lemma (in connected-components-locale) path-from-root-num:
  fixes v :: ′a
  assumes v ∈ verts G
  shows vpath (pwalk v) G ∧
         hd (pwalk v) = r ∧
         last (pwalk v) = v ∧
         num v = vwalk-length (pwalk v)
using assms
proof (induct vwalk-length (pwalk v) arbitrary: v rule: less-induct)
case less
  thus ?case
  proof (cases v=r)
    case True
      thus ?thesis using r-assms unfolding vpath-def by force
  next
    case False
      then obtain e where ee:
        e ∈ arcs G
        e = the (parent-edge v)
        head G e = v ∧ num v = num (tail G e) + 1
        using less.prems parent-num-assms by force
      let ?te = tail G e
      let ?p′ = pwalk ?te
      let ?q = [?te, v]
      obtain p where
        pp: p = ?p′ ⊕ ?q
        by presburger
      hence pv: p = pwalk v
        using less.prems False ee(2) by force
      have ew: vwalk ?q G unfolding vwalk-def
        using ee(3) in-arcs-imp-in-arcs-ends[OF ee(1)]
              less.prems  tail-in-verts[OF ee(1)]
        by auto
      have wlp: vwalk-length ?p′ < vwalk-length (pwalk v)
        using vwalk-length-pwalk[OF less.prems False] ee(2)
        by presburger
```

```
      hence pp':
        vwalk ?p' G
        distinct ?p'
        hd ?p' = r
        last ?p' = ?te
        num ?te = vwalk-length ?p'
        using less.hyps[where v=?te,
              OF - tail-in-verts[OF ee(1)]]
        unfolding vpath-def by linarith+
      have jp: joinable ?p' ?q
         unfolding joinable-def
         by (simp only: pp'(4) pp'(1)[unfolded vwalk-def], simp)
      have vwalk-length [tail G e, v] = 1 by force
      hence np: num v = vwalk-length p
        using pp vwalk-join-vwalk-length[OF jp] ee pp'(5)
        by (simp only: pp vwalk-join-vwalk-length[OF jp] ee pp'(5))
      have wp: vwalk p G
        by (metis pp ew pp'(1) jp vwalk-joinI-vwalk)
      {
        fix x assume xp: x ∈ set ?p'
        have vwalk-length (pwalk x) ≤ vwalk-length ?p'
        using pwalk-split[OF xp] by (smt length-append vwalk-length-simp)
        then have wlx: vwalk-length (pwalk x) < vwalk-length (pwalk v)
          using wlp by linarith
        hence num x = vwalk-length (pwalk x)
          using pp'(1) less.hyps[OF wlx] xp vwalk-verts-in-verts by blast
        with wlx have num x < vwalk-length (pwalk v) by presburger
      }
    then have v ∉ set ?p' using wlp np pv  by (metis less-not-refl)
    hence dp: distinct p
      by (metis butlast-snoc distinct.simps(2) distinct1-rotate pp pp'(2)
        list.simps(2) rotate1.simps(2) rotate1-hd-tl vwalk-join-def)
    hence vpath p G ∧ hd p = r ∧ last p = v ∧
          num v = vwalk-length p
      using dp wp np pp' pp
      by (metis hd-append2 last-snoc list.sel(3) pwalk-ne vpathI vwalk-join-def)
    thus ?thesis using pv by fast
  qed
qed
```

```
definition
  no-loops :: ('a, 'b) pre-digraph ⇒ bool
where
  no-loops G ≡ ∀ e ∈ arcs G. tail G e ≠ head G e
```

```
definition
  abs-IGraph :: IGraph ⇒ (32 word, 32 word) pre-digraph
where
  abs-IGraph G ≡ ⦇ verts = {0..<ivertex-cnt G}, arcs = {0..<iedge-cnt G},
    tail = fst o iedges G, head = snd o iedges G ⦈

lemma verts-absI[simp]: verts (abs-IGraph G) = {0..<ivertex-cnt G}
  and edges-absI[simp]: arcs (abs-IGraph G) = {0..<iedge-cnt G}
  and start-absI[simp]: tail (abs-IGraph G) e = fst (iedges G e)
  and target-absI[simp]: head (abs-IGraph G) e = snd (iedges G e)
  by (auto simp: abs-IGraph-def)

definition
  abs-pedge :: (32 word ⇒ 32 word) ⇒ 32 word ⇒ 32 word option
where
  abs-pedge p ≡ (λv. if sint (p v) < 0 then None else Some (p v))

lemma None-abs-pedgeI[simp]:
  ((abs-pedge p) v = None) = (sint (p v) < 0)
  using abs-pedge-def by auto

lemma Some-abs-pedgeI[simp]:
  (∃ e. (abs-pedge p) v = Some e) = (sint (p v) ≥ 0)
  using None-not-eq None-abs-pedgeI
  by (metis abs-pedge-def linorder-not-le option.simps(3))
```

```
lemma wellformed-iGraph:
  assumes wf-digraph (abs-IGraph G)
  shows ⋀e. e < iedge-cnt G ⟹
        fst (iedges G e) < ivertex-cnt G ∧
        snd (iedges G e) < ivertex-cnt G
using assms unfolding wf-digraph-def by simp

lemma path-length:
  assumes vpath p (abs-IGraph iG)
  shows vwalk-length p < unat (ivertex-cnt iG)
proof –
  have pne: p ≠ [] and dp: distinct p using assms by fast+
  have unat (ivertex-cnt iG) = card (unat ` {0..<(fst iG)})
    using unat-image-upto by simp
  then have unat (ivertex-cnt iG) = card ((verts (abs-IGraph iG)))
     by (simp add: inj-on-def card-image)
  hence length p  ≤ unat (ivertex-cnt iG)
```

```
      by (metis finite-code card-mono vwalk-def
          distinct-card[OF dp] vpath-def assms)
  hence length p − 1 < unat (ivertex-cnt iG)
    by (metis pne Nat.diff-le-self le-neq-implies-less
        less-imp-diff-less minus-eq one-neq-zero length-0-conv)
  thus vwalk-length p < unat (fst iG)
    using  assms
    unfolding vpath-def vwalk-def by simp
qed

lemma ptr-coerce-ptr-add-uint[simp]:
  ptr-coerce (p +_p uint x) =  p +_p  (uint x)
  by auto

lemma check-r′-spc:
  is-graph s iG p ⟹
   is-numm s iG iN p′ ⟹
   is-pedge s iG iP p″ ⟹
   check-r′ p r p″ p′ s =
   Some (if is-root iG r iP iN then 1 else 0)
  unfolding check-r′-def unfolding is-graph-def is-numm-def is-pedge-def
  apply (simp add: ocondition-def oguard-def ogets-def oreturn-def obind-def)
  apply (simp add: is-root-def uint-nat word-less-def sint-ucast)
  apply (safe, simp-all add: arrlist-nth)
     apply (fastforce simp: dest:arrlist-nth-value[where i=int (unat r)])
    apply (fastforce dest:arrlist-nth-valid[where i=int (unat r)])
   apply (fastforce dest:arrlist-nth-value[where i=int (unat r)])
  apply (fastforce dest:arrlist-nth-valid[where i=int (unat r)])
  done


lemma pedge-num-heap:
  ⟦arrlist (λp. heap-w32 h (ptr-coerce p)) (λp. is-valid-w32 h (ptr-coerce p))
  (map (iL ∘ of-nat) [0..<unat n]) l; i < n⟧ ⟹
    iL i = heap-w32 h (l +_p int (unat i))
  apply (subgoal-tac
  heap-w32 h (l +_p int (unat i)) = map (iL ∘ of-nat) [0..<unat n] ! unat i)
   apply (subgoal-tac map (iL ∘ of-nat) [0..<unat n] ! unat i = iL i)
    apply fastforce
   apply (metis (hide-lams, mono-tags) unat-mono word-unat.Rep-inverse
    minus-nat.diff-0 nth-map-upt o-apply plus-nat.add-0)
  apply (simp add: arrlist-nth-value unat-mono)
  done

lemma pedge-num-heap-ptr-coerce:
  ⟦arrlist (λp. heap-w32 h (ptr-coerce p)) (λp. is-valid-w32 h (ptr-coerce p))
```

```
(map (iL ∘ of-nat) [0..<unat n]) l; i < n; 0 ≤ i⟧ ⟹
  iL i = heap-w32 h (ptr-coerce (l +p int (unat i)))
apply (subgoal-tac
heap-w32 h (ptr-coerce (l +p int (unat i))) = map (iL ∘ of-nat) [0..<unat n] ! unat i)
 apply (subgoal-tac map (iL ∘ of-nat) [0..<unat n] ! unat i = iL i)
  apply fastforce
 apply (metis (hide-lams, mono-tags) unat-mono word-unat.Rep-inverse
  minus-nat.diff-0 nth-map-upt o-apply plus-nat.add-0)
apply (drule arrlist-nth-value[where i=int (unat i)], (simp add:unat-mono)+)
done
```

```
lemma edge-heap:
  ⟦ arrlist h v (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ep;
  e < m⟧ ⟹ to-edge ((iedges iG) e) = h (ep +p (int (unat e)))
  apply (subgoal-tac h (ep +p (int (unat e))) =
  (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ! unat e)
   apply (subgoal-tac to-edge ((iedges iG) e) =
   (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ! unat e)
    apply presburger
   apply (metis (hide-lams, mono-tags) length-map length-upt o-apply
      map-upt-eq-vals-D minus-nat.diff-0 unat-mono word-unat.Rep-inverse)
  apply (fastforce simp: unat-mono arrlist-nth-value)
  done
```

```
lemma head-heap:
  ⟦arrlist h v (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ep; e < m⟧ ⟹
  snd ((iedges iG) e) = t-C (h (ep +p (uint e)))
  using edge-heap to-edge.simps t-C-pte by (metis uint-nat)
```

```
lemma tail-heap:
  ⟦arrlist h v (map (to-edge ∘ (iedges iG ∘ of-nat)) [0..<unat m]) ep; e < m⟧ ⟹
  fst ((iedges iG) e) =  s-C (h (ep +p  (uint e)))
  using edge-heap to-edge.simps s-C-pte uint-nat by metis
```

```
lemma check-parent-num-spc':
  ⦃ P and
    (λs. wf-digraph (abs-IGraph iG) ∧
         is-graph s iG g ∧
         is-numm s iG iN n ∧
         is-pedge s iG iP p ∧
         r < ivertex-cnt iG)⦄
  check-parent-num' g r p n
  ⦃ (λ- s. P s) And
    (λrr s. rr ≠ 0 ⟷ parent-num-assms-inv iG r iP iN (unat (ivertex-cnt iG))) ⦄!
```

```
apply (clarsimp simp: check-parent-num'-def)
apply (subst whileLoopE-add-inv[where
      M=λ(vv, s). unat (ivertex-cnt iG − vv) and
      I=λvv s. P s ∧ parent-num-assms-inv iG r iP iN (unat vv) ∧
        vv ≤ ivertex-cnt iG ∧
        wf-digraph (abs-IGraph iG) ∧
        is-graph s iG g ∧ is-numm s iG iN n ∧
        is-pedge s iG iP p ∧
        r < ivertex-cnt iG])
apply (simp add: skipE-def)
apply wp
  unfolding is-graph-def is-numm-def is-pedge-def parent-num-assms-inv-def
  apply (subst if-bool-eq-conj)+
  apply (simp split: split-if-asm, safe, simp-all add: arrlist-nth)
                              apply (rule-tac i= (uint vv) in arrlist-nth-valid, simp+)
                              apply (metis uint-nat word-less-def)
                             apply (rule-tac x=unat vv in exI)
                             apply (subgoal-tac n-C (heap-Graph-C s g) ≤ iN vv)
                              apply (metis (hide-lams) word-less-nat-alt
                              word-not-le word-unat.Rep-inverse)
                             apply (subst pedge-num-heap[where l=n and iL=iN])
                               apply simp
                              apply simp
                             apply (metis uint-nat)
                            apply (rule-tac i= (uint vv) in arrlist-nth-valid)
                              apply simp+
                            apply (metis uint-nat word-less-def)
                           apply (rule-tac x=unat vv in exI)
                           apply (rule conjI, metis unat-mono, simp)
                           apply (metis sint-ucast not-le uint-nat
                           pedge-num-heap-ptr-coerce word-zero-le)
                          apply (rule-tac x=unat vv in exI)
                          apply (rule conjI, metis unat-mono, simp)
                      apply (metis not-le uint-nat pedge-num-heap-ptr-coerce word-zero-le)
                         apply (rule-tac x=unat vv in exI)
                         apply (rule conjI, metis unat-mono, simp)
                         apply (subgoal-tac snd (snd (snd iG) (iP vv)) =
                          t-C (heap-Edge-C s (es-C (heap-Graph-C s g) +_p uint (iP vv))))
                          apply (metis uint-nat pedge-num-heap-ptr-coerce word-zero-le)
                         apply (subst head-heap[where iG=iG], simp)
                      apply (metis not-le uint-nat pedge-num-heap-ptr-coerce word-zero-le)
                         apply simp
                        apply (rule-tac x=unat vv in exI)
                        apply (rule conjI, metis unat-mono, simp,clarsimp)
                        apply (subgoal-tac iN vv ≠ iN (fst (snd (snd iG) (iP vv))) + 1)
                         apply fast
```

```
         apply (subst pedge-num-heap[where l=n and iL=iN])
           apply simp+
         apply (subst pedge-num-heap[where l=n and iL=iN])
           apply simp
          apply (drule wellformed-iGraph[where G=iG])
           apply simp+
         apply (subst tail-heap[where iG=iG], simp+)
         apply (subst pedge-num-heap-ptr-coerce[where l=p and iL=iP])
            apply simp+
         apply (metis uint-nat)
        apply (drule less-unat-plus1, safe, blast)
        apply (subst pedge-num-heap-ptr-coerce[where l=p and iL=iP])
           apply simp+
        apply (metis sint-ucast not-less uint-nat)
       apply (drule less-unat-plus1, safe, blast)
       apply (subst pedge-num-heap-ptr-coerce[where l=p and iL=iP])
          apply simp+
       apply (metis not-less uint-nat)
      apply (drule less-unat-plus1, safe, blast)
      apply (subst pedge-num-heap-ptr-coerce[where l=p and iL=iP])
         apply simp+
      apply (subst head-heap[where iG=iG], (simp add: uint-nat)+)
     apply (drule less-unat-plus1, safe, blast)
     apply (subst pedge-num-heap[where l=n and iL=iN], simp+)
     apply (subst pedge-num-heap[where l=n and iL=iN], simp)
      apply (drule-tac e=iP vv in  wellformed-iGraph[where G=iG])
       apply (metis not-le pedge-num-heap-ptr-coerce word-zero-le)
      apply simp
     apply (subst tail-heap[where iG=iG], simp+)
      apply (metis not-le pedge-num-heap-ptr-coerce word-zero-le)
     apply (subst pedge-num-heap-ptr-coerce[where l=p and iL=iP])
        apply simp+
     apply (metis uint-nat)
    apply (drule less-unat-plus1, safe, blast)
    apply (subst pedge-num-heap[where l=n and iL=iN])
      apply (simp add: uint-nat)+
   apply (metis le-def word-le-nat-alt word-not-le
   less-unat-plus1 eq-of-nat-conv)
  apply (metis unat-minus-plus1-less)
 apply (rule arrlist-nth, blast, blast)
 apply (simp add: uint-nat unat-mono)
apply (rule arrlist-nth, blast, blast)
apply (simp add: uint-nat)
   apply (drule-tac i=vv in pedge-num-heap-ptr-coerce[where l=p and
iL=iP])
  apply simp+
```

```
               apply (drule-tac e=iP vv in wellformed-iGraph[where G=iG])
                apply simp+
               apply (drule-tac e=iP vv in tail-heap[where iG=iG])
                apply (simp add: uint-nat unat-mono)+
              apply (rule arrlist-nth, (simp add: uint-nat unat-mono)+)+
          apply (metis less-unat-plus1 word-unat.Rep-inverse)
         apply (metis eq-of-nat-conv less-unat-plus1)
        apply (metis (hide-lams, no-types) eq-of-nat-conv less-unat-plus1)
       apply (metis (no-types) less-unat-plus1 word-unat.Rep-inverse)
      apply (metis (no-types) less-unat-plus1 word-unat.Rep-inverse)
     apply (metis inc-le)
    apply (metis unat-minus-plus1-less)
   apply metis
  apply wp
  apply fast
  done
lemma num-less-n:
  fixes v
  assumes is-root G r p n
  assumes parent-num-assms-inv G r p n (unat (ivertex-cnt G))
  assumes v < ivertex-cnt G
  shows n v < ivertex-cnt G
proof -
  have ivertex-cnt G > 0
    using assms by (metis word-neq-0-conv word-not-simps(1))
  thus ?thesis
    using assms unfolding parent-num-assms-inv-def is-root-def
    by (cases v=r, presburger , metis unat-mono word-unat.Rep-inverse)
qed

lemma parent-num-assms-inv-num-ne-0:
  fixes v
  assumes wf-digraph (abs-IGraph G)
  assumes is-root G r p n
  assumes parent-num-assms-inv G r p n (unat (ivertex-cnt G))
  assumes v ≠ r
  assumes v < (ivertex-cnt G)
  shows n v ≠ 0
proof–
  have p v ∈ arcs (abs-IGraph G)
    using assms(3–5) unat-mono
    unfolding parent-num-assms-inv-def
    by fastforce
  hence fst (iedges G (p v)) ∈ verts (abs-IGraph G)
     using assms(1) wf-digraph-def by fastforce
  hence n (fst (snd (snd G) (p v))) < ivertex-cnt G
```

```
    using num-less-n[OF assms(2,3)] by fastforce
  moreover
  have n v = n (fst (snd (snd G) (p v))) + 1
    using assms unat-mono
    unfolding parent-num-assms-inv-def
    by force
  ultimately
  show ?thesis using assms
  by (metis less-is-non-zero-p1)
qed

lemma connected-components-locale-num-eq-invariants':
⋀G r p n.
  (connected-components-locale (abs-IGraph G) (unat ∘  n) (abs-pedge p) r
  ∧ (∀ v ∈ verts (abs-IGraph G). v≠ r ⟶ (unat ∘ n) v < unat (ivertex-cnt G))) =
    (wf-digraph (abs-IGraph G) ∧
    is-root G r p n ∧
    parent-num-assms-inv G r p n (unat (ivertex-cnt G)))
proof –
  fix G fix r::32 word fix p n::32 word ⇒ 32 word
  let ?aG = abs-IGraph G
  let ?ap = abs-pedge p
  let ?an = unat ∘  n
  let ?wf = wf-digraph ?aG
  let ?is-root = r ∈ verts ?aG ∧ ?ap r = None ∧ ?an r = 0
  let ?pnai = (∀ v. v ∈ verts ?aG ∧ v ≠ r ⟶
                  (∃ e∈arcs ?aG. ?ap v = Some e ∧
                  head ?aG e = v ∧
                  ?an v = ?an (tail ?aG e) + 1)) ∧
               (∀ v. v ∈verts ?aG ∧  v ≠ r ⟶
                  ?an v < unat (ivertex-cnt G))
  have isr-eq: ?is-root = is-root G r p n
    unfolding is-root-def
    using None-abs-pedgeI unat-eq-0 by auto
moreover
  have (?wf ∧ ?is-root ∧ ?pnai)
      = (?wf ∧ is-root G r p n ∧
        parent-num-assms-inv G r p n (unat (ivertex-cnt G)))
  proof –
  {
    assume wf: ?wf
    assume isr: ?is-root
    assume ∗: ⋀ v. v ∈ verts ?aG ∧ v ≠ r ⟹
    (∃ e ∈ arcs ?aG. ?ap v = Some e ∧
    head ?aG e = v ∧
    ?an v =  ?an (tail ?aG e) + 1) ∧ (?an v < unat (ivertex-cnt G))
```

```
  {
    fix i
    let ?i = of-nat i
    assume i < unat (ivertex-cnt G) ∧ ?i ≠ r
    then have ii: ?i ∈ verts (abs-IGraph G) ∧ ?i ≠ r
      by (simp add: word-of-nat-less)
    then obtain e where e-assms:
      e ∈ arcs ?aG
      ?ap ?i = Some e
      head ?aG e = ?i
      ?an ?i =  ?an (tail ?aG e) + 1
      ?an ?i < unat (ivertex-cnt G) using ∗[OF ii] by auto
    have pi-e: p ?i = e
      using e-assms(2) abs-pedge-def Some-abs-pedgeI
      by (cases ?ap ?i) force+
    with e-assms pi-e  Some-abs-pedgeI have
      p ?i < iedge-cnt G ∧
       0 ≤ sint (p ?i) ∧
       snd (iedges G (p ?i)) = ?i ∧
       n ?i = n (fst (iedges G (p ?i))) + 1 ∧
       n ?i < ivertex-cnt G ∧
       n ?i ≠ 0
       by (auto,
           metis Some-abs-pedgeI,
           metis (hide-lams, mono-tags) Suc-eq-plus1 unat-1
                 word-arith-nat-add word-unat.Rep-inverse,
           metis word-less-nat-alt)
  } then have is-root G r p n ∧
              parent-num-assms-inv G r p n (unat (ivertex-cnt G))
  unfolding parent-num-assms-inv-def using isr isr-eq by blast
}
hence ?wf ∧ ?is-root ∧ ?pnai
    ⟹ is-root G r p n ∧
      parent-num-assms-inv G r p n (unat (ivertex-cnt G)) by presburger
moreover
{
  assume wf: ?wf
  assume isr: is-root G r p n
  assume pna: parent-num-assms-inv G r p n (unat (ivertex-cnt G))
  {
    fix v
    assume vG: v ∈ verts ?aG
    assume vnr: v ≠ r
    have uvG: unat v < unat (ivertex-cnt G)
      using vG by (simp add: word-less-nat-alt)
    have nv-ne0: n v ≠ 0 using pna isr wf unfolding parent-num-assms-inv-def
```

```
        by (metis parent-num-assms-inv-num-ne-0 pna uvG vnr word-less-nat-alt)
      then have ∗:
        p v < iedge-cnt G ∧
        0 ≤ sint (p v) ∧
        snd (iedges G (p v)) = v ∧
        n v = n (fst (iedges G (p v))) + 1 ∧
        n v < ivertex-cnt G
        using vnr pna
        unfolding parent-num-assms-inv-def
        by (metis uvG word-unat.Rep-inverse)
      then have 1:
      ∃ e. e ∈ arcs ?aG ∧ ?ap v = Some e ∧
          head ?aG e = v ∧
          ?an v = ?an (tail ?aG e) + 1
        using abs-pedge-def linorder-not-less unatSuc2 nv-ne0 by auto
     have 2: ?an v < unat (ivertex-cnt G)
     using ∗ by (metis o-apply word-less-nat-alt)
     from 1 2 have
     (∃ e. e ∈ arcs ?aG ∧ ?ap v = Some e ∧
          head ?aG e = v ∧
          ?an v = ?an (tail ?aG e) + 1) ∧
      ?an v < unat (ivertex-cnt G) by blast
   } then have ?is-root ∧ ?pnai using isr isr-eq by fast
  }
  hence ?wf ∧ is-root G r p n ∧
        parent-num-assms-inv G r p n (unat (ivertex-cnt G)) ⟹
        ?is-root ∧ ?pnai by presburger
  ultimately
    show ?thesis by blast
  qed
ultimately
show ?thesis G r p n
  unfolding connected-components-locale-def
  connected-components-locale-axioms-def
  fin-digraph-def fin-digraph-axioms-def
  by auto
qed

lemma cc-num-less-n:
  assumes connected-components-locale (abs-IGraph G) (unat ∘ n) (abs-pedge p) r
  assumes v ∈ verts (abs-IGraph G)
  shows (unat ∘ n) v < unat (ivertex-cnt G)
using connected-components-locale.path-from-root-num[OF assms] path-length
by presburger

lemma connected-components-locale-eq-invariants′:
```

```
⋀G r p n.
  (connected-components-locale (abs-IGraph G) (unat ∘  n) (abs-pedge p) r) =
    (wf-digraph (abs-IGraph G) ∧
    is-root G r p n ∧
    parent-num-assms-inv G r p n (unat (ivertex-cnt G)))
    using connected-components-locale-num-eq-invariants′ cc-num-less-n by blast

lemma check-connected-spc:
  ⦃ P and
    (λs. wf-digraph (abs-IGraph iG) ∧
          is-graph s iG g ∧
          is-numm s iG iN n ∧
          is-pedge s iG iP p)⦄
  check-connected′ g r p n
  ⦃ (λ- s. P s) And
    (λrr s. rr ≠ 0 ⟷
      connected-components-locale (abs-IGraph iG) (unat ∘  iN) (abs-pedge iP) r) ⦄!
  apply (clarsimp simp: check-connected′-def
  connected-components-locale-eq-invariants′)
  apply wp
  apply (rule-tac P1= P and
    (λs. wf-digraph (abs-IGraph iG) ∧
          is-graph s iG g ∧
          is-numm s iG iN n ∧
          is-pedge s iG iP p ∧
          r < ivertex-cnt iG ∧
          is-root iG r iP iN)
     in validNF-post-imp[OF - check-parent-num-spc′])
  unfolding fin-digraph-def fin-digraph-axioms-def
  apply force
  apply wp
  apply (auto simp: check-r′-spc is-root-def)[]
done

end
end
```

Curriculum Vitae: Christine Rizkallah

Personal Details

Birth date	27.01.1987
Address (work)	Campus E1 4, 66123 Saarbrücken, Germany
E-Mail	crizkall@mpi-inf.mpg.de
Webpage	http://www.mpi-inf.mpg.de/users/crizkall

Education

Since 07.2010	**PhD Student at Max Planck Institute for Informatics (MPII), Saarbrücken, Germany** Research: *Certifying Algorithms, Theorem Proving* Advisor: Prof. Kurt Mehlhorn Group: Algorithms and Complexity
10.2007 - 12.2009	**MSc in Computer Science at Saarland University, Saarbrücken, Germany** Final Grade: 1.6 (B^+) Advisors: Dr. Chad E. Brown, Prof. Gert Smolka Thesis: *Proof Representations for Higher Order Logic*
10.2003 - 10.2007	**Bachelor (B.S.) in Computer Science and Engineering at German University in Cairo, Egypt** Final Grade: 1.39 (A^-) Thesis: *Hierarchial Task Network Planning for Real Time Applications*
10.2001 - 07.2003	**International General Certificate of Secondary Education (IGCSE), Dar El Tarbeya, Cairo, Egypt** Final Grade: 108% (A^+)

Scholarships

2007 - 2009	IMPRS-CS Fellowship for Master studies in Computer Science. Max-Planck Institute for Informatics, Saarbrücken, Germany

2007	Full coverage of tuition fees for the 8th semester at the German University in Cairo due to ranking first in class (out of 66 students) in the 7th semester.
2003 - 2007	Category *A* (reduced) tuition fees at the German University in Cairo due to high performance in IGCSE.
2011-2012	Received travel fund for attendance from the following: - FLOC 2014, Vienna - MOD 2013, Marktoberdorf - OPLSS 2012, Oregon, Portland - NASSLLI 2012, Austin, Texas - Algorithmic Frontiers Workshop 2012, Lausanne - Microsoft Research PhD Summer School 2011, Cambridge - CodeF 2011, Google, Munich

Publications

Authors ordered alphabetically, else clarified in paper.

Click on **bold** paper names to get redirected to papers.

C. E. Brown and C. Rizkallah. **Glivenko and Kuroda for Simple Type Theory**. In *Journal of Symbolic Logic (JSL)*. Submitted 2011. To appear 2014.

C. Rizkallah. **A Simpl Shortest Path Checker Verification**. In *Isabelle Workshop* at *ITP*. To appear 2014.

L. Noschinski, C. Rizkallah, and K. Mehlhorn. **Verification of Certifying Computations through AutoCorres and Simpl**. In *NASA Formal Methods (NFM)* 2014, volume 8430 of LNCS, pages 46-61.

E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. **A Framework for the Verification of Certifying Computations**. In *Journal of Automated Reasoning (JAR)* 2014, volume 52(3), pages 241-273. Preliminary version appeared under the name **Verification of Certifying Computations**. In *Computer Aided Verfication (CAV)* 2011, volume 6806 of LNCS, pages 418-423.

C. E. Brown and C. Rizkallah. **From Classical Extenional Higher-Order Tableau to Intuitionistic Intensional Natural Deduction**. *Proof Exchange for Theorem Proving (PxTP)* at *CADE*, pages 27-42. 2013.

C. Rizkallah. **An Axiomatic Characterization of the Single-Source Shortest Path Problem**. In: Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, The Archive of Formal Proofs. May 2013.

C. Rizkallah. **Maximum Cardinality Matching**. In: Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, The Archive of Formal Proofs. July 2011.

E. Alkassar, S. Böhme, K. Mehlhorn, C. Rizkallah, and P. Schweizer. **An Introduction to Certifying Algorithms**. *it - Information Technology* 2011, volume 53, pages 287-293.

Work Experience

11.2013 - 4.2014	**Trustworthy Systems, NICTA, Sydney, Australia** *Intern*, Verification of file systems (team member).
Since 07.2010	**Algorithms and Complexity, Max-Planck Institut für Informatik, Saarbrücken, Germany** *Doctoral student*
11.2009 - 12.2009	**Information and Technology Management, Saarland University, Saarbrücken, Germany** *Full time*, Development of Java bootstrapping alg.
4.2007 - 8.2007	**Artificial Intelligence, Xaitment GmbH (a spin-off of DFKI), Saarbrücken, Germany** *Intern*, Development of C++ planner (B.S. thesis).
8.2006 -10.2006	**Software Development, Owita GmbH, Lemgo, Germany** *Intern*, Development of J2ME application for testing and installing Bluetooth based Sensor-Adaptor-Modules.
7.2006 - 8.2006	**Software Development, Cotton Refinement Co., Cairo, Egypt**

	Intern, Designing their inventory system (team member).
7.2005 - 8.2005	**IT Department, PetroJet, Cairo, Egypt** *Intern*, Development of a database system for new applicants (team member).

Teaching

Winter 2012/2013	*Seminar on Social Choice Theory*, Saarland University, lecturer (together with Rob van Stee).
Summer 2011	*Graph Theory*, Saarland University, teaching assistant.
Winter 2005/2006	*Data Structures and Algorithms*, German University in Cairo, junior teaching assistant.
Summer 2005	*Introduction to Java*, German University in Cairo, junior teaching assistant.

Other Academic Activities

Since 8.2010	Member of the IMPRS PhD application committee
Since 8.2010	PhD Representative of the Algorithms group
Since 9.2010	PhD Representative of MPI für Informatik
2013	Reviewer for the Computational Geometry journal (CGTA)
2011	Co-organizer of Max Planck Advanced Course on the Foundations of Computer Science (ADFOCS 2011)
2011	Co-organizer of the PhDnet Interdisciplinary Event 2011
2010	Reviewer for Information Processing Letters (IPL)

Talks, Poster Presentations, and Attended Conferences

Talks	Isabelle workshop 2014, NFM 2014, MOD 2013 (student session), POPL 2013 (student session), OPLSS 2012 (student session), EPFL 2011, MPI 2009, MPI 2010, MPI 2011, MPI 2013
Posters	Algorithmic Frontiers 2012, Microsoft Research summer school 2011, MPI 2011, MPI 2012
Attended	CAV 2014, ITP 2014, NFM 2014, MOD 2013, PLMW 2013, POPL 2013, OPLSS 2012, NASSLI 2012, Algorithmic Frontiers 2012, VTSA 2012, Personality based

communication workshop 2012, PhDnet meeting 2012, CodeF 2011, CAV 2011, Microsoft Research summer school 2011, ADFOCS 2011, PhDnet scientific event 2011, ADFOCS 2010, PhDnet meeting 2010, Team management and conflict resolution workshop 2010.

Language Skills

Arabic(mother tongue), English(fluent), German(intermediate), Spanish(intermediate).

Saarbrücken, September 22, 2015