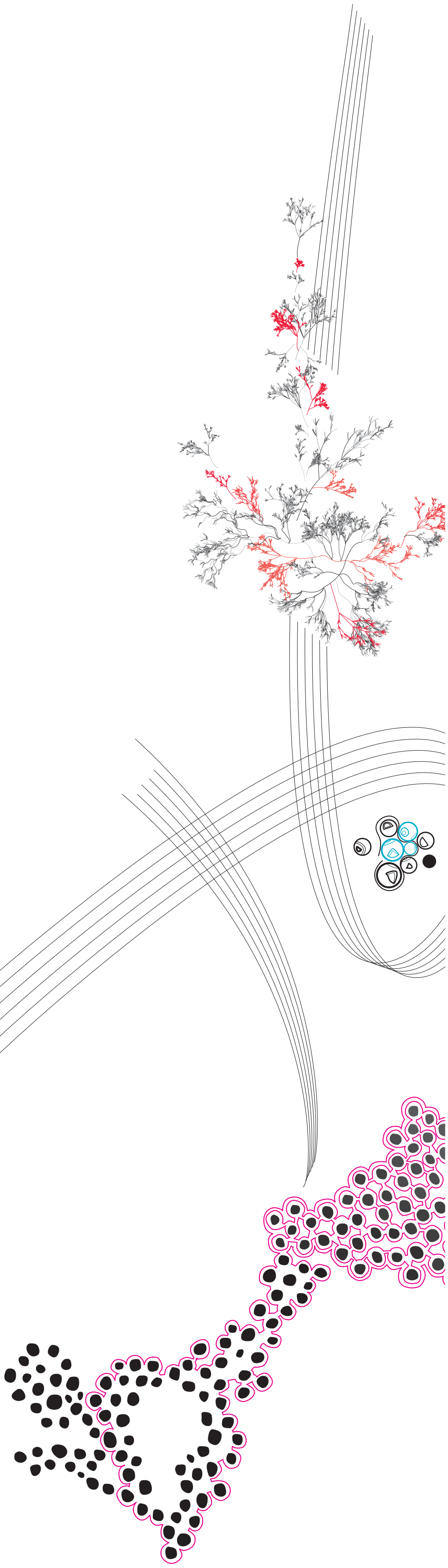
Formal Verification of a Sequential SCC Algorithm



Johannes Gerrit Jan Boerman

February, 2023

Committee:
Prof.Dr. M. Huisman
R.B. Rubbens, MSc
Prof.Dr. C.E.W. Hesselman

Formal Methods and Tools research group
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Abstract

Correctness of software is becoming an increasingly important subject in today's age where software becomes more and more ubiquitous. As opposed to software testing where software is executed to test correct behaviour in only a limited number of scenarios, we apply deductive verification - a technique which allows us to prove with certainty that the software is programmed correctly for all possible scenarios. In this thesis we zoom in on a Strongly Connected Components algorithm that is used inside model checkers. We formally verify a set of correctness properties that should be satisfied by the algorithm. To achieve this we use the deductive verifier VerCors which is able to statically verify assertions formulated using first-order logic formulas. We discuss our approach and proofs, and reflect on the limitations and issues that were encountered in the process. We also discuss how these issues could be addressed. Lastly, we briefly touch on how our approach would need to be adjusted to verify the parallel version of this algorithm.

Acknowledgements

I would like to give some thanks to a handful of people who helped me out in the process of doing this research. Firstly, I would like to think my supervisor Marieke Huisman for offering me a desk at the FMT research group. Working in an office with people with similar research interests boosted my morale noticeably. Secondly I would like to thank Pieter Bos, Lukas Armborst, Bob Rubbens, Ömer Şakar, Petra van den Bos and Sophie Lathouwers for answering any VerCors-related questions I had. Thirdly, I would like to thank the people from graduation support groups with whom I've had tremendously useful weekly conversations that kept me motivated. In particular I would like to name Lambert Forkink, Antonio Sanchez Martin, Ramon Houtsma, Cornelis ten Napel, Erik Kemp, Daan Kooij, Ivan Hop, Bas Bleijerveld and Wouter Bos. Thanks to all of you!

Contents

1. Introduction

1.1 Motivation

Strongly Connected Components (SCCs) are a well-established topic in computer science, with algorithms for computing them existing already since the 1970s. There are path-based SCC algorithms such as Tarjan's algorithm [25], Munro's algorithm [17] or Gabow's algorithm[8]. Other types of SCC algorithms are *forward-backward* algorithms and *nested depth-first search*. Strongly Connected Components, also known as *transitive closure*s are subsets of vertices in a graph that are all connected to each other via a *path*. For a more formal definition, please refer to Section 2.2.

SCCs often find their use as an intermediate step in a larger calculation. For example the importance of web pages for web search results can be determined in a distributed setting using SCCs of web pages that link to each other [5]. Another use case is LTL Model Checking where SCCs are used to compute subsets of a labeled transition systems with *accepting cycles*, explained in more detail in Section 2.3. Because the model checker itself is used to verify safety properties for models, it is of utmost importance that the model checker itself is correct. This gives reason to formally verify the SCC algorithm that is used inside of it.

In this thesis we will be verifying the sequential SCC algorithm presented in the PhD thesis of Vincent Bloemen [2], this is a path-based SCC algorithm. To verify correctness properties of the algorithm, we will make use of the deductive verification tool VerCors, developed at the University of Twente [3].

1.2 Problem statement & Goals

Bloemen presents in his PhD thesis both a sequential SCC algorithm, as well as a multi-core SCC algorithm. The sequential SCC algorithm uses concepts already well-established from literature and Bloemen presents an informal proof for the correctness. The parallel SCC algorithm (*UFSCC*) is his own invention where he uses a new concurrent union-find (Section 3.1) datastructure that is efficient for checking whether two vertices are part of the same partition, and for iterating over vertices that are not yet completely explored. Bloemen also presents a correctness intuition for the parallel algorithm. Because the algorithm is used inside the LTSmin [16] model checking toolset, we want to be absolutely sure about the correctness, which gives reason to machine-check the correctness proof. As a start we want to formally verify correctness properties of the sequential algorithm. Formalising and verifying the parallel algorithm is left as future research.

1.3 Preliminary work

We build or proofs for the correctness of the sequential SCC algorithm on the work of Johannes P. Hollander [11]. His work presents an encoding of the Bloemen's pseudo code into PVL, one of the programming languages supported by VerCors. In his work he already verifies all data structures of the algorithm, including the union-find data structure that he implements using a sequence. He also verifies properties that state which vertices are present in the data structures, and suggests PVL-definitions for other (unverified) correctness properties. Later on in his thesis, he compares VerCors to other deductive verification tools, and gives recommendations to the VerCors team to make VerCors more suitable for the verification of graph algorithms.

1.4 Contribution

We continue Hollander's work by proving two additional properties of the algorithm:

- We fill one of the holes left in Hollander's formalisation; at one point in the algorithm Hollanders *assumes* that the parameter v is present in the *Live* set - but never verifies this assumption. We present a proof for this assumption in Section 3.4.4 and verify it using VerCors.
- Additionally, we verify the fact that this SCC algorithm correctly produces *fitting strongly-connected* components (Definition 2) in Section 3.4.5.

Ultimately, there is still extra work required to verify that algorithm produces *maximal* SCCs (Definition 3), but we are certain that this property is also satisfied. All source code and verification code from this thesis is available online at https://github.com/Jankoekenpan/VerCors. The SCC algorithm code is located in the examples/scc directory.

1.5 Research questions

This thesis will answer the following research questions:

RQ1. What are the techniques needed to verify the correctness of a high level graph algorithm?

RQ2. What are the obstacles when verifying a high level graph algorithm?

1.6 Thesis structure

This thesis is organised as follows:

- Chapter 2 contains background information on deductive verification, VerCors' architecture and LTL model checking, and all formalities involved. Additionally three types of SCCs are introduced.
- Chapter 3 discusses the details of the SCC algorithm. It also discusses the approach to proving correctness. A multitude of properties and invariants are discussed, as well as their formalisations.
- Chapter 4 contains the results of our verification work. A dependency graph of properties is presented, and verification times are discussed.
- In Chapter 6 we interpret the verification results and draw conclusions. We also reflect on the verification experience of VerCors, and give some tips & tricks.
- Chapter 7 describes where to go next using these results.
- Chapter 5 describes how our work compares to other work that has been done in the field.

2. Background

In this chapter we discuss the required background information. We will discus the background theory for deductive verification (Section 2.1) and Strongly Connected Components (SCCs) (Section 2.2). We also discuss how SCCs are used in LTL model checking (Section 2.3).

2.1 Deductive verification

Deductive verification is the art of proving properties about programs using a deductive logic. This section discusses the theoretical background for this. In Section 2.1.2 we discuss how VerCors takes these concepts from theory to practice.

2.1.1 Floyd-Hoare logic

The concept of deducing new facts after execution of some program statements was formalised by Tony Hoare [9] and separately by Robert Floyd [7]. Hoare introduces so-called *Hoare triples*. Whilst Hoare initially wrote them as $\mathcal{P}\{Q\}\mathcal{R}$, we use the updated notation $\{\mathcal{P}\}S\{\mathcal{R}\}$. In this notation S represents a statement (or: *command*) from a programming language. $\mathcal{P}$ represents the set of facts that are true before S is executed (*precondition*). $\mathcal{R}$ is the set of facts that are true after S finished executing (*postcondition*). This is often referred to as *partial correctness*, meaning we cannot make any guarantees when S does not terminate.

Rules

Hoare defined multiple rules for different types of statements in an Algol-like programming language. The rules allow us to reason about programs by stating facts that are true before and after execution of a command. By applying the rules using a proof tree we can verify a program consisting of multiple statements.

- Assignment axiom

$$\frac{}{\vdash \{P_0\}x := f\{P\}}\ {\scriptstyle Ass}$$

  Here x is a variable identifier, f is an expression, P is a predicate that is satisfied after the assignment, and P_0 is obtained from P by substituting all occurrences of x by f. Consequently we obtain $P_0(f) = P(x)$: f in environment P_0 is equal to x in environment P.

- Composition rule:

$$\frac{\vdash \{P\}S_1\{Q\} \qquad \vdash \{Q\}S_2\{R\}}{\vdash \{P\}S_1; S_2\{R\}}\ {\scriptstyle Comp}$$

  Here S_1 and S_2 denote two programs, and $S_1; S_2$ denotes the sequential composition of these programs: 'S_1 followed by S_2'. When P holds before S_1 and Q after S_1, and when Q holds before S_2 and R after S_2, then we can conclude that the composed programs guarantees that R holds afterwards, given P beforehand. One can intuitively explain this using the following informal notation: $\{P\}S_1\{Q\}S_2\{R\}$.

- Consequence rules:

$$\frac{\vdash \{P\}S\{Q\} \qquad \vdash Q \rightarrow R}{\vdash \{P\}S\{R\}}\,{\scriptstyle Cons_1} \qquad \frac{\vdash \{Q\}S\{R\} \qquad \vdash P \rightarrow Q}{\vdash \{P\}S\{R\}}\,{\scriptstyle Cons_2}$$

  The consequence rules formalise the notion of postcondition weakening and precondition strengthening. Rule (1) means: 'If program S has precondition P and postcondition Q, then any (weaker) proposition R that is logically implied by Q is also a postcondition of S.' Similarly for (2), if Q is a precondition of S, then any (stronger) proposition P that implies Q is also a precondition of S.

- Iteration rule:

$$\frac{\vdash \{P \wedge B\}S\{P\}}{\vdash \{P\}\texttt{while}\ B\ \texttt{do}\ S\{\neg B \wedge P\}}\,{\scriptstyle Iter}$$

  In this example we call P the *loop invariant*; it holds both before and after the loop. B is the *loop condition*. If a statement S requires B and preserves P, then this rule allows us to prove that P still holds after executing S repeatedly. After the loop is finished we can assume $\neg B$ because at that point the loop condition must be false.

Using these rules we verify the following example program:

```
int i = 0;
while (i < 10) {
    i = i + 1;
}
assert i == 10;
```

This can be formalised using the following Hoare triple:

$$\{true\}i := 0; \texttt{while}\ \ i < u\ \ \texttt{do}\ \ i := i + 1\{i = 10\}$$

.

The complete proof tree is written on the right hand side of the page. We read the proof from bottom to top.

1. First we apply the composition rule (*Comp*) in order to split the statements $i := 0$ and $\texttt{while}\ \ i < 10\ \ \texttt{do}\ \ i := i + 1$. The left hand side then then trivially proven using the assignment axiom (*Ass*), but the right hand-side requires more work still.

2. Using the second consequence rule ($Cons_2$) we massage the precondition into a weaker form ($0 \leq i \leq 10$), that will serve as our loop invariant.

3. Next up, we also write our postcondition in terms of the loop invariant and the inverse of the loop condition (*loop invariant:* $0 \leq i \leq 10$).

4. Then we apply the iteration rule (*Iter*) which leaves us with just having to prove preservation of the loop invariant.

5. This is trivially proven by applying the first consequence rule ($Cons_1$) for postcondition weakening, and then applying the assignment axiom (*Ass*) for the remaining triple. □

$$\dfrac{\dfrac{}{\vdash \{0 \leq i < 10\}i := i + 1\{0 < i \leq 10\}}\,Ass \qquad \vdash 0 < i \rightarrow 0 \leq i}{\vdash \{0 \leq i \leq 10 \wedge i < 10\}i := i + 1\{0 \leq i \leq 10\}}\,Cons_1$$

$$\dfrac{\vdash \{0 \leq i \leq 10 \wedge i < 10\}i := i + 1\{0 \leq i \leq 10\}}{\vdash \{0 \leq i \leq 10\}\texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{0 \leq i \leq 10 \wedge \neg(i < 10)\}}\,Iter$$

$$\dfrac{\vdash \{0 \leq i \leq 10\}\texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{0 \leq i \leq 10 \wedge \neg(i < 10)\}}{\vdash \{0 \leq i \leq 10\}\texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{i = 10\}}\,\textit{loop invariant: } 0 \leq i \leq 10$$

$$\dfrac{\vdash i = 0 \rightarrow 0 \leq i \leq 10 \qquad \vdash \{0 \leq i \leq 10\}\texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{i = 10\}}{\vdash \{i = 0\}\texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{i = 10\}}\,Cons_2$$

$$\dfrac{\dfrac{}{\vdash \{true\}i := 0\{i = 0\}}\,Ass \qquad \vdash \{i = 0\}\texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{i = 10\}}{\vdash \{true\}i := 0; \texttt{while}\ i < 10\ \texttt{do}\ i := i + 1\{i = 10\}}\,Comp$$

2.1.2 VerCors architecture

In this section we take a look at the VerCors tool set [3], its architecture, and its frond-end for verification engineers.

Overview

VerCors is a static verifier and can be conceptualised as one big tree transformer. Figure 2.1 shows its high-level architecture. VerCors accepts input languages Java, OpenCL, OpenMP and PVL. PVL stands for *Prototypical Verification Language* and is specifically designed for VerCors. PVL is an object oriented language much like Java, but without advanced concepts such as inheritance. PVL also natively supports the axiomatic data types of VerCors [21] such as *set*, *sequence* and *map*. Programs written in these input languages can be annotated with formal specifications. These specifications come in the form of assert statements, loop invariants, and contracts and they will be further explained in Subsection 2.1.2.

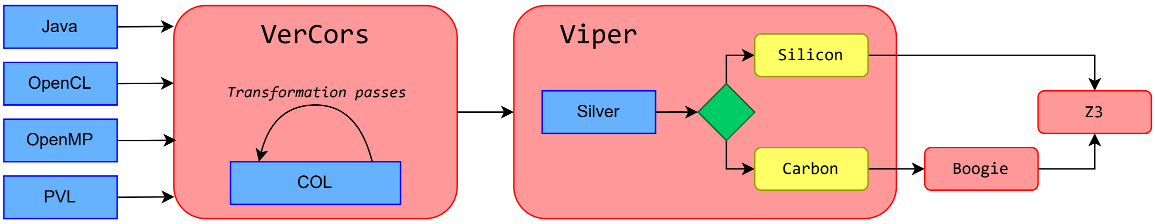


Figure 2.1: VerCors' high level architecture

VerCors parses input programs into COL, *Common Object Language*, the intermediate representation used by VerCors. From there, additional passes are applied to the COL tree until it is transformed into a tree in the Silver language. Silver is the input language for Viper, the back-end of VerCors. Viper is developed at ETH in Zürich and the team there maintains two verifiers that can verify a Silver program [24]. These are Silicon [23] and Carbon [6]. Silicon makes use of symbolic execution whereas Carbon uses verification condition generation and then calls into Boogie [4]. Both Silicon and Carbon (eventually) translate the Silver tree into an SMT problem (Satisfiability Modulo Theories), which is then checked by the Z3 theorem prover [27]. Both Boogie and Z3 are developed by Microsoft Research. In this thesis we will be using the Silicon/Z3 combination for verification. Once verification is complete, VerCors will output either of the following results:

- Pass - The program code was verified to adhere to the formal specifications.
- Fail - The program could not be verified. (At least) one of the specifications could not be proven to hold. In this situation VerCors will always output which specification could not be verified.

For now we will neglect the third option which is: VerCors does not terminate within a reasonable amount of time. Possible solutions for this issue are discussed in Section 6.1.

VerCors specification language

The VerCors specification language is inspired by JML [13] which follows the *Design By Contract* philosophy. In this chapter we will discuss the specifications used in this case study. We start out with two examples, in Listing 2.1 and 2.2. In Listing 2.1 at line 3 an integer is declared with value

5. VerCors can verify this fact, using the assertion at line 4. VerCors also understands addition; when 2 is added to x, then at line 6 VerCors can verify that the value of x is then 7.

Listing 2.1: AddAssignJava.java example

```
1 class AddAssignJava {
2     void test() {
3         int x = 5;
4         //@ assert x == 5;
5         x += 2;
6         //@ assert x == 7;
7     }
8 }
```

Listing 2.2: Loop.pvl example

```
1  class Counter {
2    int x;
3  }
4
5  class Loop {
6      requires c != null ** Perm(c.x,1);
7      requires y >= 0;
8      ensures  c != null ** Perm(c.x,1);
9      ensures  c.x == \old(c.x) + y;
10     void incr(Counter c, int y) {
11         int i = 0; l
12         loop_invariant 0 <= i && i <= y;
13         loop_invariant c != null ** Perm(c.x, 1);
14         loop_invariant c.x == \old(c.x) + i;
15         while (i < y) {
16             c.x = c.x + 1;
17             i = i + 1;
18         }
19     }
20 }
```

The Loop.pvl example is more involved, it shows the usage of contracts and loop invariants, as well as permissions. We shall explain these concepts line by line and explain some important keywords.

Lines	**Explanation**
Lines 1-3	This is a data carrier class containing an integer. It will be used in the method INCR in the Loop class.
Lines 6-7	These are the *preconditions* of the INCR method. The `Counter` argument passed to c must not be null, and the caller must have write access to the field x of c. Additionally the argument passed to y must be non-negative. A caller must satisfy these conditions before INCR can be called. The preconditions are then assumed to be true at the start of the body (line 11).

Lines 8-9	These are the *postconditions*. These statements are guaranteed by the method, so the programmer must ensure that these statements hold at every exit point of the method. This method contains no return statements, so there is only one exit point (line 18). In this case the method guarantees that c is still non-null, and $c.x$ has been incremented by y amount. The caller also gets full write permission to $c.x$. A caller can assume that these statements are true after INCR returns.
Lines 12-14	These are the *loop invariants*. Loop invariants are used to prove that some properties hold throughout the entire loop. VerCors checks that the loop invariants hold before the loop starts (establishment), and also at the end of the loop body (preservation). Note that line 12 contains $i \leq y$, but the while condition only specifies $i < y$. A loop invariant of $i < y$ would not be valid here, since at line 17 in the last iteration i becomes the same value as y.

Keyword	**Explanation**
`**`	This is the 'separating conjunction' operator. One can think of it as similar to `&&`, but it differs in the fact that `**` imposes that no aliasing takes place (i.e. variables used in operands refer to different memory locations). Whilst this intuition is not completely correct, it suffices for the purposes for this research. Further details on separating conjunction can be found in [20]. We pronounce '`a ** b`' as '*a* and separately *b*'.
`Perm`	`Perm(`*location*`,` $fraction$`)` means that the routine requires permission to read or write to a variable. The fraction is a value in range [0..1] and it written as *numerator**denominator*. The value '1' means that the routine has full write permission, whereas a value between 0 and 1 indicates a read-only permission. The value '0' itself indicates 'no access'.
`\old`	`\old(`*location*`)` refers to the value of *location* before the method start. It is most useful in postconditions and loop invariants.

Listing 2.2 shows a naive addition algorithm. It adds the value of y to $c.x$, by incrementing $c.x$ repeatedly by 1 (y many times). The loop invariant at line 14 states that before and after every iteration the value of $c.x$ equals the sum of the old value of $c.x$ and i. When the loop terminates VerCors knows that $0 \leq i \wedge i \leq y \wedge \neg(i < y)$ hence it can conclude $i = y$, and thus the postcondition $c.x = \backslash old(c.x) + y$ can be proven by taking the loop invariant and substituting i for y. Below are some more keywords listed which will be used later in this research.

- `\result`: Can be used in postconditions. Refers to the return value of a method or function.
- `context`: This is a combination of `requires` and `ensures`. '`requires` *a*' and '`ensures` *a*' can be replaced by '`context` *a*'.
- `context_everywhere`: Copies the provided boolean expression to `requires`, `ensures` and `loop_invariant` clauses.
- `\forall`: (`\forall` *declaration; condition; expression*) allows us to specify that some expression is true for some range of values. For example: `(\forall int i; 0 <= i && i < arr.length; arr[i] >= 0)` states that all elements in the array *arr* are non-negative.

The type of both the *condition* and the *expression* must be boolean. This concept is otherwise known as 'universal quantification'.

- `\exists`: (`\exists` *declaration; condition; expression*) Rather than stating some property holds 'for all' values, this states that there must 'exist' some value, satisfying the condition and expression. This is otherwise known as 'existential quantification'.
- `==>` denotes implication: 'a `==>` b' has the same meaning as $a \to b$. The expression only evaluates to `true` when b is true, or a and b have the same value.
- `pure`: Modifier for methods and functions (but concrete functions are implicitly pure already). `pure` means that the method has no side-effects, i.e. it does not assign to fields.
- `inline`: `inline` is a function modifier as well. Inline functions are not called, but instead their body is inserted at the call site in one of the transformation passes of VerCors. The `inline` keyword serves as a useful technique to assign names to invariants.
- `given`: `given` is used in contracts. It provides a way to add extra (ghost) parameters that are needed for verification of methods.
- `yields`: `yields` is the dual of `given`. It provides a way to add extra (ghost) return values to methods. One can also think of them as out-parameters.
- `static`: `static` has the same meaning as `static` in Java. When a method or function is declared as `static`, it is not dependent on the instance object; instead there is no dynamic dispatch and the method only depends on the class.
- `assume`: The statement `assume` a causes the verifier to assume that a holds at that point in the program. Expressions that are *assumed* in this manner are not verified, hence it is possible to introduce knowledge that contradicts pre-existing knowledge, leading to unsoundness. The `assume` keyword should therefore only used in parts of the program that have yet to be verified.
- Triggers: function calls in quantified expressions can be surrounded with extra colons and curly braces to guide VerCors to the completion of the proof. For example, if one can assert `(\forall int i; {:f(i):}; g(i))` and then asserts `f(5)`, the trigger would get instantiated and `g(5)` is added to the knowledge base. More information about triggers can be found on the VerCors wiki [29].

This is not a complete list, but it provides a nice basis for understanding our formalised proofs later on in Section 3.4.

2.2 Strongly Connected Components

In this section we discuss the theory around Strongly Connected Components (SCCs). Bloemen describes three types of SCCs [2]. In a graph $G = (V, E)$ we say that $v \rightarrow v' \in E$ is an edge. A path is a sequence of vertices $v_1 \cdot \ldots \cdot v_N$ where all subsequent pairs of vertices $\ldots \cdot v_i \cdot v_{i+1} \cdot \ldots$ are connected by an edge: $(v_i, v_{i+1}) \in E$. A path from x to y consisting of multiple edges is denoted as $x \rightarrow^* y$. We write $x \leftrightarrow y$ iff $x \rightarrow^* y \wedge y \rightarrow^* x$ and call x and y *strongly connected*. Then we use the following definitions:

Definition 1. *A Partial Strongly Connected Component (PSCC) is a set of vertices $C \subseteq V$ for which all pairs of nodes are strongly connected, i.e.*

$$PSCC(C) \triangleq \forall x, y \in C.\ x \leftrightarrow y$$

Definition 2. *A Fitting Strongly Connected Component (FSCC) is a PSCC with the additional requirement that at least one of the paths has all vertices contained within C. i.e.*

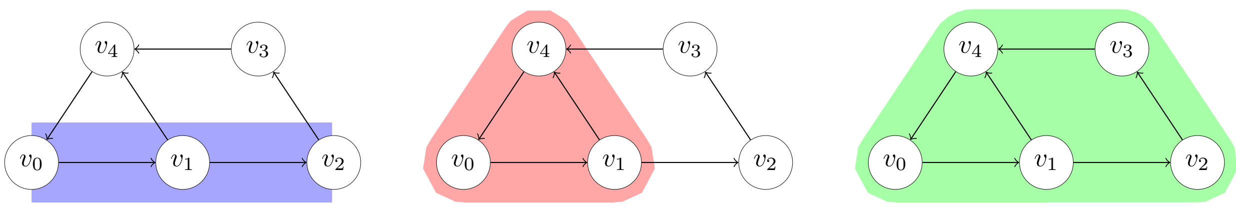
$$FSCC(C) \triangleq PSCC(C) \wedge \exists p\, s.t.\, p[0] \in C \wedge p[|p| - 1] \in C.\ \forall i \in 0 \ldots |p|.\ p[i] \in C$$

Definition 3. *A maximal Strongly Connected Component (SCC) is an FSCC with the additional requirement that there exists no other vertex $z \notin C$ that is strongly connected to some vertex in C. i.e.*

$$SCC(C) \triangleq FSCC(C) \wedge \neg\exists z \in V \wedge z \notin C.\ \exists x \in C.\ x \leftrightarrow z$$

.

Figure 2.2 examplifies these definitions. In Figure 2.2a $\{v_0, v_1, v_2\}$ is a PSCC because $v_0 \leftrightarrow v_1 \wedge v_0 \leftrightarrow v_2 \wedge v_1 \leftrightarrow v_2$. Note that for PSCCs it is not required that the paths are completely inside of the marked region. In Figure 2.2b $\{v_0, v_1, v_4\}$ is an FSCC because all pairs of vertices in the region are connected by a path that is completely inside of the region. For FSCCs is it not required that every vertex that 'could be added while still retaining strong connectivity' is part of the set. The green region in Figure 2.2c is an SCC because it is an FSCC which has no outgoing paths that reaches back into it.



(a) *PSCC* (b) *FSCC* (c) *SCC*

Figure 2.2: Three types of SCCs

2.3 LTL model checking

Model checking is the art of modelling a complex system as a transition system, and then specifying correctness properties using a formal logic. A model checking tool then checks whether the model adheres to the specification. The result can be either *Pass* (in which case the model satisfies the specification), or *Fail* (in which case the model checker always provides a counterexample for the specification that was violated). Several flavours of formal logics exists, but we focus on *linear-time temporal logic* (LTL) since it presents itself as a nice application for SCCs (Section 2.2) [18]. LTL formulas [14] enable us to specify correctness properties over possibly infinitely long running systems. We make the distinction between *safety* and *liveness* properties:

- Safety: 'something bad never occurs'
- Liveness: 'something good will eventually happen'

Examples of safety properties are: 'The railway barriers are never open when a train passes', or (in the context of computer programs) 'Variable x is never `null`'. Examples of liveness properties are: 'The sun will eventually rise' or 'The program will eventually terminate'. A model checker answers the question: 'Does model $\mathcal{M}$ satisfy property ϕ?' Formally we write this as $\mathcal{M} \models \phi$ where $\mathcal{M}$ is the model, and ϕ is the (LTL) formula. To produce an answer, the model checker performs the following (high level) steps (also shown in Figure 2.3):

1. Generate state-space of model ($A_{\mathcal{M}}$), negate the LTL formula and convert to Büchi automaton ($A_{\neg\phi}$).
2. Synchronise the two automata into a product.
3. Check whether the language of the product is empty. (This is where SCCs are involved, see Paragraphs 2.3.1)
4. If so, then $\mathcal{M} \models \phi$.
5. If not, then the model checker finds a word that is accepted by both $A_{\mathcal{M}}$ and $A_{\neg\phi}$. This run is the *counterexample* of ϕ.

In the following subsections we will discuss each step and explain the definitions.

2.3.1 Kripke Structures & Büchi Automata

In LTL model checking, Kripke Structures are used to model complex systems. Kripke Structures can be generated from the model description. We refer to these Kripke Structures as the 'Model Automata'. Some model checkers employ an optimisation where they do not need to generate the entire state space up-front, instead they lazily generate only the parts that are required for the LTL formula. We call this 'on-the-fly' model checking [1].

Whereas the models are transformed into Kripke Structures, the LTL formulas get negated and then transformed into Büchi Automata [1]. Büchi automata are essentially a special type of Kripke Structures that are labeled with accepting marks. These marks can be positioned either at the states, or at the transitions. Typical for Büchi automata is that they can accept infinite inputs, where each infinite input string causes the automaton to visit all accepting marks infinitely often. In order for this to happen, all accepting marks must be present in a cycle that is reachable by a prefix

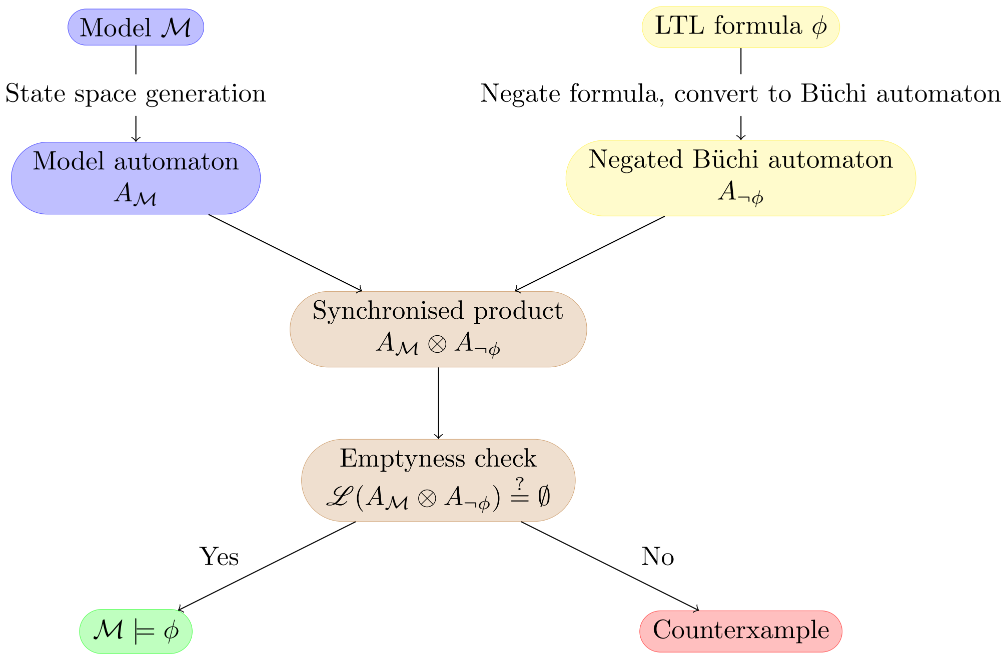


Figure 2.3: LTL Model checking process

of the input string. The rest of the input string must be of repeating nature. To find such cycles, model checkers employ SCC-finding algorithms. The intrigued reader can continue this section to read more about LTL model checking, but it is not necessary to understand the main contents of this thesis in Chapter 3.

Kripke Structures

Kripke Structures are graphs whose vertices represent possible states of a system, and whose edges represent state transitions. Kripke Structures also contain a marking functions which assigns a set of propositions to each state. Formally we write $M = (S, S_0, R, AP, L)$ where:

- S is the set of all states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the *total* transition relation. It is guaranteed that $\forall_{s \in S} \exists_{t \in S} R(s, t)$.
- AP is the set of atomic propositions.
- $L : S \to \mathcal{P}(AP)$ is the labelling function that assigns a set of labels to each state.

Büchi Automata

Büchi Automata (BA) come in four forms: TGBA, SGBA, TBA, SBA: *Transition-based Generalised Büchi Automaton*, *State-based Generalised Büchi Automaton*, *Transition-based Büchi Automaton* and *State-based Büchi Automaton*. All four forms are equivalent in expressiveness, and algorithms exist to convert from one form to the others. Barnet et al.[1] define a TGBA as a 5-tuple $A = (Q, \iota, \delta, n, M)$ where:

- Q is a finite set of states,
- $\iota \in Q$ is the initial state,
- $\delta \subseteq Q \times \mathbb{B}^{AP} \times Q$ is a set of transitions,
- n is an integer specifying the number of accepting marks,
- $M : \delta \to \mathcal{P}([n])$ is a marking function that specifies a subset of marks associated with each transition. (N.B. $[n]$ denotes the set of non-negative integers until n, i.e. $[n] = \{0, 1, \ldots, n-2, n-1\}$.

A TBA is a specialised TGBA where $n = 1$. SGBA is defined similarly as TGBA, except that the marking function M assigns marks to states instead of transitions. For SGBA the type of M is $Q \to \mathcal{P}([n])$. Consequently, an SBA is a Büchi automaton with only one acceptance mark, which is assigned to just one of the states.

Once the Model Automaton and negated formula BA are combined, then the model checker converts the combined automaton into a TBA or SBA and starts the search for accepting cycles. Note that combining the automata can also be done 'on-the-fly'.

Synchronised product

This paragraph explains the *synchronised product* of two automata. This is the operation that combines the Model Automaton ($A_{\mathcal{M}}$) and Büchi Automaton ($A_{\neg\phi}$) into one: $A_{\mathcal{M}} \otimes A_{\neg\phi}$.

Let $K = (S_1, \iota_1, R_1, AP, L_1)$ be a Kripke structure with one initial state, and $A = (Q_2, \iota_2, \delta_2, n, M_2)$ a TGBA. The synchronised product $K \otimes A$ is then a TGBA, defined as $(Q', \iota', \delta', n, M')$ where

- $Q' = S_1 \times Q_2$,
- $\iota' = (\iota_1, \iota_2)$,
- $((s_1, s_2), x, (d_1, d_2)) \in \delta' \iff (s_1, d_1) \in R_1 \wedge L_1(s_1) = x \wedge (s_2, x, d_2) \in \delta_2$,
- $M'(((s_1, s_2), x, (d_1, d_2))) = M((s_2, x, d_2))$.

One way to intuitively think about this product is a parallel composition of the Kripke Structure and the Büchi automaton:

- The resulting set of states Q' is the carthesian product of the states from both sources.
- The resulting initial state ι' is the product of both initial states.
- The resulting set of transitions δ' features only those transitions that have their source-destination pairs in R_1, as well as in δ_2. The labels of the transitions are required to be present on s_1 in K.
- The resulting labeling function M' labels transitions when M_2 labels the states from Q_2.

The emptiness-check problem

Step 3 specifies that the problem of checking whether a language is empty can be reduced to the problem of checking for *accepting cycle*s. Namely, a word is only present in the language if all acceptance marks of the automaton are visited infinitely often. Hence, these acceptance marks are required to be present in a cycle that is reachable from the initial state. We also say that accepting runs are *lasso*-shaped.

Runs, Words and Languages

A *run* ρ through a Büchi automaton is a sequence of transitions, where the source state of the first transition is ι, and all the destination states equal the source states of the next transition. Bernat et. al [1] defines the set of runs accepted by the automaton as:

$$Runs(A) = \{\, \rho \in \delta^\omega \mid \rho(0)^s = \iota \wedge \forall i \geq 0.\, \rho(i)^d = \rho(i+1)^s \,\}$$

. where we denote the source of a transition t as t^s, the label as t^ℓ, and the destination as t^d. The ω indicates that the set of runs can contain infinite sequences of transitions.

The set of *accepting runs* are those runs that visit all accepting marks *infinitely often*:

$$Acc(A) = \{\, \rho \in Runs(A) \mid [n] = \bigcup_{t \in Inf(\rho)} M(t) \,\}$$

For SGBA we obtain:

$$Acc(A) = \{\, \rho \in Runs(A) \mid [n] = \bigcup_{t \in Inf(\rho)} M(t^s) \,\}$$

A *word* $\ell(\rho)$ associated with run ρ is defined using $\ell(\rho)(i) = \rho(i)^\ell$, i.e. a word is the concatenation all the labels of the transitions in a run. The *language* of a Büchi automaton is the set of words associated with all accepting runs:

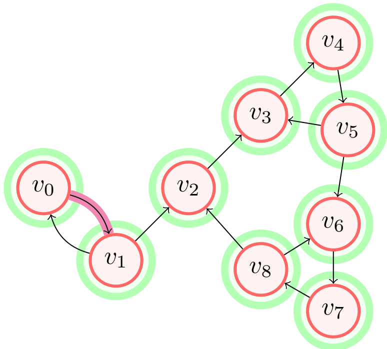
$$\mathscr{L}(A) = \{\, \ell(\rho) \mid \rho \in Acc(A) \,\}$$

It then follows that the words accepted by $K \otimes A$ are the words accepted by both K and A, i.e. $\mathscr{L}(K \otimes A) = \mathscr{L}(K) \cap \mathscr{L}(A)$. If the model checker finds that $\mathscr{L}(A_{\mathcal{M}} \otimes A_{\neg\phi}) = \emptyset$, then that means that no input exists that is accepted through $\neg\phi$, hence all possible runs of $A_{\mathcal{M}}$ satisfy ϕ!

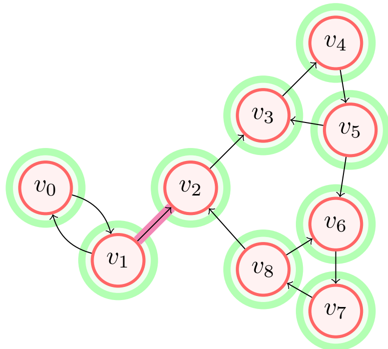
3. Set-Based SCC Algorithm

In this chapter we will discuss an algorithm that finds SCCs in directed graphs. The algorithm was originally invented by Munro [17] and later updated by Bloemen [2]. We will define correctness properties to verify, and discuss the details of the proofs and formalisations they use.

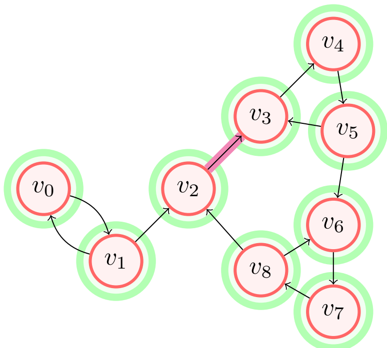
An example execution of the algorithm can be seen in figure 3.2. When the algorithm terminates, it has found all the SCCs reachable from the starting node. In this example, it finds both SCCs.



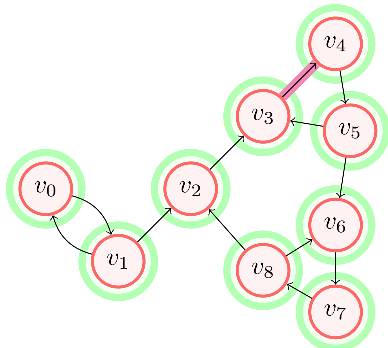
1. SetBased(v_0) $\rightarrow v_1$



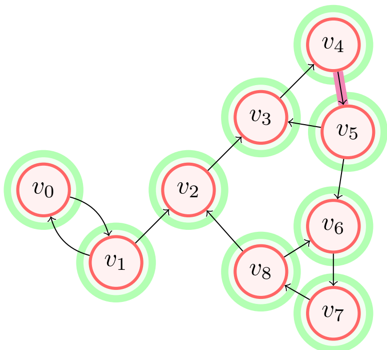
2. SetBased(v_1) $\rightarrow v_2$



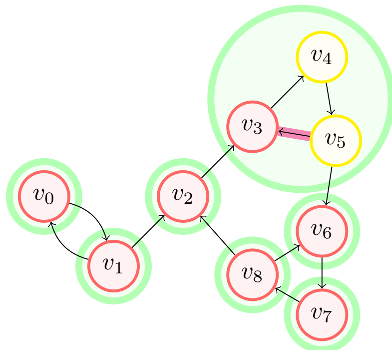
3. SetBased(v_2) $\rightarrow v_3$



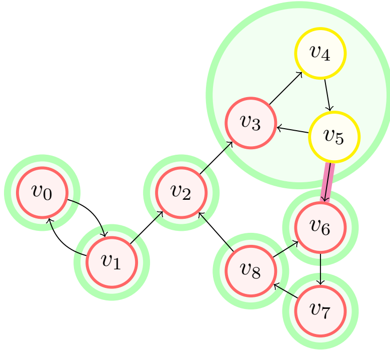
4. SetBased(v_3) $\rightarrow v_4$



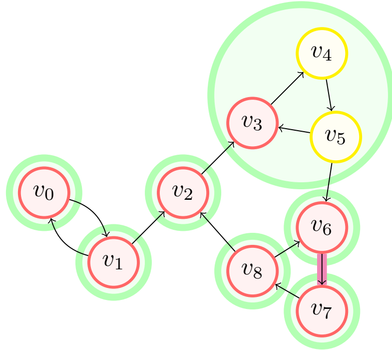
5. SetBased(v_4) $\rightarrow v_5$



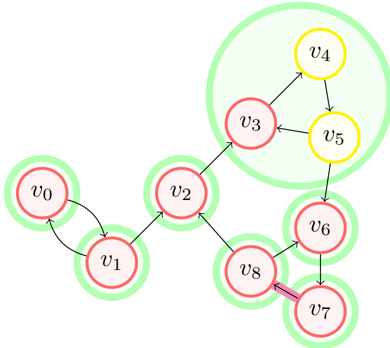
6. SetBased(v_5) $\rightarrow v_3$



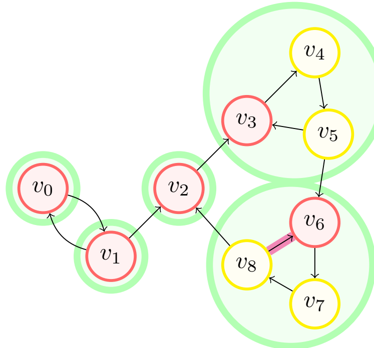
7. SetBased(v_5) $\rightarrow v_6$



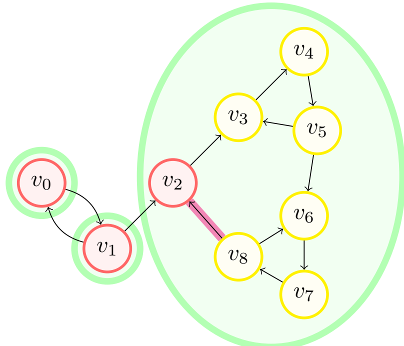
8. SetBased(v_6) $\rightarrow v_7$



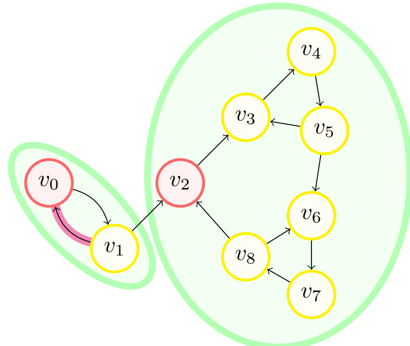
9. SetBased(v_7) $\rightarrow v_8$



10. SetBased(v_8) $\rightarrow v_6$



11. SetBased(v_8) $\rightarrow v_2$



12. SetBased(v_1) $\rightarrow v_0$

Figure 3.2: Set-based SCC algorithm example execution

In each subfigure a new successor is visited, and the state of the data structures used is updated. In subfigure (1) we start out at v_0 and visit successor v_1. In subfigure (2) the algorithm has a choice, it could first visit v_0 again, or it could visit v_2. In this example it visits successor v_2. Then in subfigure (3) it goes to v_3, to v_4 in (4) and v_5 in (5). Now in subfigure (6) the algorithm visits v_3 again, meaning that all edges of the cycle have been found. The green areas represent groups of vertices that are known to be reachable from one another, so the three green circles are collapsed into one. We call such a group of vertices highlighted in green a *partition*. This same process happens after we visit v_6, v_7 and v_8 in subfigure (10). At step (11) the algorithm finds that v_8 reaches v_2, meaning that the previously collapsed partition are now themselves collapsed into one even larger partition. Lastly, the algorithm backtracks again and finds the last successor of v_1, which is v_0, and merges their partitions as well.

Observations

We make the following observations:

1. At its core, the algorithm is a depth-first-search (DFS) algorithm that keeps track of which vertices have already been seen in order to avoid searching forever.
2. Every vertex starts out in its own partition.
3. There is only one red labelled vertex per partition. The earliest visited red vertex of a partition always stays red. We call the red vertices *representatives* of their partitions. When a vertex is coloured yellow, it means that it is not special in any sense.
4. Throughout execution of the algorithm, all vertices in a partition have a path to all other vertices in that partition. These paths stay within the partition, thus partitions remain strongly connected.
5. When the algorithm is terminated, all found partitions are maximal SCCs (with respect to the set of reachable vertices).

These observations will form the basis for the formalised proof later on in Section 3.4.

3.1 Union-Find

To efficiently store the partitions we use a union-find data structure. Conceptually, a union-find data structure is a set of sets. Each inner set is a *partition*, and it is disjoint from all other partitions in the data structure. Each such partition has one special element: the *representative*. We represent the union-find with a forest of trees where the roots point to themselves. Each tree corresponds to one partition in the union-find, and the root of the tree is the representative. An example union-find is visualised in Figure 3.3. Since every node must have a parent pointer, the root of the tree points to itself. We can re-arrange this forest in a linear structure, where nodes are ordered from low to high.

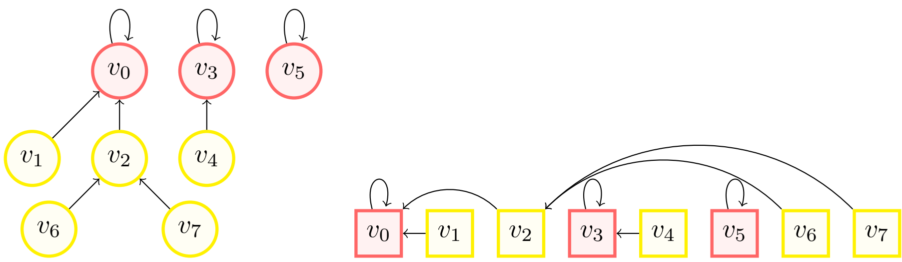


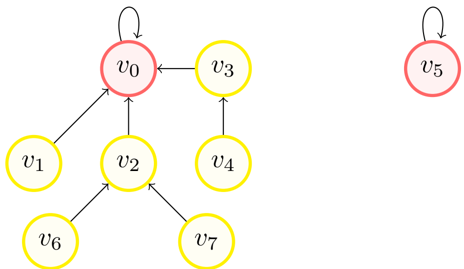
Figure 3.3: Union-Find example visualisations

From the right arrangement it becomes clear that a union-find forest can be implemented simply using a sequence of integers, where each value equals the (0-based) index of its parent. For our example union-find this is shown in Table 3.1. This method of implementing a union-find is described by Bloemen [2].

0	0	0	3	3	5	2	2

Table 3.1: Union-Find sequence

We denote our union-find value with the letter S. To find the representative of a node, one simply walks the chain of parents, until a node points to itself. For the representative of v_2 we can write $S.rep(v_2) = S.rep(v_1) = S.rep(v_0) = v_0$. To find the partition set of a node, we collect all other nodes that share the same representative, i.e. $S.part(x) \triangleq \{\, v_i \mid i \in 0 \ldots N \wedge S.rep(i) = S.rep(x) \,\}$. We otherwise write $S.part(x)$ simply as $S(x)$. In this example v_0 represents $\{v_0, v_1, v_2, v_6, v_7\}$, v_3 represents $\{v_3, v_4\}$ and v_5 represents only itself. Uniting two partitions a and b can be done by having the representative of a point to the representative of b. This is a cheap operation since only one value needs to be updated in the sequence. It is constant-time when the representatives of the partitions are already known, since then there is no need to traverse the trees in order to find them. Uniting $S(v_0)$ and $S(v_3)$ gives us:



(a) Tree representation

0	0	0	0	3	5	2	2

(b) Sequence representation

Figure 3.4: Union-find with $S(v_0)$ united with $S(v_3)$.

In this example v_0 has become the parent of v_3, but it would have worked just as well if v_3 became the parent of v_0. In both cases, the united partition remains a tree where all nodes are represented by the same node.

3.2 Pseudo code

The pseudo code is adapted for verification and listed in Listing 3.1. From this listing the constant and variable declarations are omitted. They are as follows:

Constants:

- G is the graph.

Variables:

- $Visited$ is the set of visited vertices, it grows with every recursive call.

- *Explored* is the set of nodes that are visited and also all their descendants are visited. The *Explored* set only grows per found SCC.
- S is the union-find structure, starting out with every vertex being its own representative. Partitions are united as the algorithm finds cycles.
- R is the stack of roots. It supports the *push*, *pop* and *top* operations. It contains only nodes that are representatives in S. All elements in R are unique.
- O is not part of the original algorithm by Bloemen [2]; it is added for verification of the algorithm, it is not required for the SCC computation. O represents the encounter order of nodes. It can be thought of as a sequence that always contains the same elements as the $Visited$ set. The only difference is that sets are unordered, but elements in O are ordered by encounter order. Further explanation on how O is used can be found in Section 3.4.4.
- H is an auxiliary variable as well. It is a sequence of vertices and represents the operand stack of the algorithm as it executes. An element is appended to H with every function call (line 4), and that same element is removed from H again right before the function returns (line 24). H contains only unique elements, and all elements in H are also in $Visited$. R is always a subsequence of H, which is further explained in Section 3.4.5.

Listing 3.1: Instrumented SCC algorithm

```
1  function SetBased(v) {
2      Visited := Visited ∪ {v};
3      O := O · v;
4      H := H · v;
5      R.Push(v);
6      for each w ∈ Succ(v) {
7          if (w ∉ Explored) {
8              if (w ∉ Visited) {
9                  SetBased(w);
10             } else {
11                 while (S(v) ≠ S(w)) {
12                     r := R.Pop();
13                     S.Unite(r, R.Top());
14                 }
15                 report FSCC S(v)
16             }
17         }
18     }
19     if (v = R.Top()) {
20         report SCC S(v);
21         Explored := Explored ∪ S(v);
22         R.Pop();
23     }
24     H := init(H);
25 }
```

3.2.1 Intuition & Correctness

The algorithm starts its exploration at an initial node v_0 and calls the SetBased function which then represents it as input variable v. At line 2-5 v is added to the $Visited$ set, encounter order O, call stack H and root stack R. It then loops over all the successors of v. We skip over all the vertices that are already *Explored* (line 7). If a successor w is unvisited, the SetBased function is called recursively in order to explore it (line 8-9) depth-first. By induction, we now have that all descendants of w are $Visited$. If the successor w was already visited, we visit the stack collapsing loop (lines 11-14). Because w is already visited, we know that there must already exist some path from w to v. (v, w) is thus a back-edge. Since v is reachable from w and w from v, v and w are part of the same SCC. The loop merges partitions until v and w are in the same partition. Additionally a root is popped from R in each iteration, ensuring that the new representative of the merged partition remains on top of the stack. After the loop finishes v and w share the same representative, which is then the top element of R. This guarantees that after the loop we can assert that the partition that contains v is an FSCC (line 15). Once the successor loop terminates we check whether v is the top element of R, and if so, it means that v is its own representative. This also means that the partition of v was not united with an older partition. We mark the partition of v as a maximal SCC and add it to the *Explored* set (lines 20-21). The intuition for $S(v)$ being a maximal SCC is that there cannot be another node that reaches $S(v)$ and is reachable from $S(v)$ since all the successors of v have been visited, and thus their partitions have been merged with $S(v)$ if they were part of the same SCC. Afterwards, v is popped from R (line 22) because it is no longer needed. Then finally, at line 24, we reassign H to the previous value of H with v removed from the end. When we return to the caller then H is equal to the call stack of the caller.

3.3 Implementation

Since the algorithm uses sets, sequences and a stack, it is best to choose an input language which understands these concepts. We choose the input language PVL since it supports operations on these data types out of the box and it is also the input language used by Hollander [11]. This section explains the translation of the concepts into PVL. In all of the code listing in this subsection the contracts are omitted for brevity. We add one global constant N to the program, and it denotes the number of vertices of the graph: $N = |V|$. One may note that a multitude of functions takes N as a parameter, without it being used in the function body. The reason for it being present is its use in the contracts. Usually its purpose is to indicate that all element in the collection are between 0 (inclusive) and N (exclusive).

Graph

We represent the graph G using an adjacency-matrix encoding (seq<seq<boolean>>). When vertex a has an edge to vertex b then $G[a][b] = true$, otherwise $false$.

Union-Find

We implement the Union-Find as a forest of trees, represented as a sequence of integers. All values in the sequence are between 0 (inclusive) and N (exclusive). We can follow a path through the union-find by treating every value as an index that is pointed to. Once a value points to its own location, then it is a root of a tree; this node is the representative. In PVL we implement the 'rep' function as described in Listing 3.2.

Listing 3.2: The 'rep' function

```
static pure int rep(int N, seq<int> S, int v) =
    S[v] == v ? v : rep(N, S, S[v]);
```

To unite two partitions, we simply have to update the pointer of one representative to point to the other representative. In PVL we could write it as in Listing 3.3. This illustrates the concept clearly, but the implementation used for verification is slightly modified. The reason for that is additional lemmas were needed to prove the postcondition of the contract. It can be found in Appendix A.

Listing 3.3: Simplified 'unite' function

```
static pure seq<int> unite(int N, seq<int> S, int v, int w) =
    S[rep(N, S, w) -> rep(N, S, v)];
```

In order to implement the function that collects all vertices of a partition, we can simply select all elements that are represented by the same representative: $part(N, S, x) = \{\, y \mid y \in [0..N) \wedge rep(N, S, y) = rep(N, S, x) \,\}$. Since PVL supports set-comprehensions it seems reasonable to use this feature, but VerCors-1.4.0 contains a bug [10] that prevents this from working when function parameters are used in the comprehension. Hence we implement it using recursion:

Listing 3.4: The 'part' function

```
static pure set<int> part(int N, seq<int> S, int v) =
    partHelper(N, S, v, 0);
static pure set<int> partHelper(int N, seq<int> S, int v,
    int i) = i < N
        ? (\let set<int> tailSet = partHelper(N, S, v, i+1);
        rep(N, S, i) == rep(N, S, v) ? {i} + tailSet : tailSet)
        : set<int> {};
```

Stack of roots

We implement the stack functions using PVL's built-in operations for slicing and indexing into sequences.

Listing 3.5: Stack functions

```
static pure seq<int> push(int N, seq<int> R, int v) =
    R ++ v;

static pure int top(int N, seq<int> R) =
   R[|R| - 1];

static pure tuple<seq<int>, int> pop(int N, seq<int> R) =
    tuple<seq<int>, int> {R[0..|R| - 1], R[|R| - 1]};
```

We note that that the bottom element of the stack is at position 0, and the top element of the stack is at the last position in the sequence. We otherwise denote the top element of the stack as R_{top}. We also note that the 'pop' function returns a tuple - the first element is the remaining stack, the second element is the top element that was removed.

Encounter order

We implement O using a map data structure instead of a sequence. This enables to implement 'indexOf' using a simple lookup, instead of a linear search. This provides benefits for verification, discussed in Section 6.1. Appending an item onto the order is done via a 'put' operation. The value of the key-value pair is the number of vertices that were already encountered before.

Listing 3.6: Encounter order functions

```
static pure map<int, int> append(map<int, int> O, int v) =
    buildMap(O, v, |O|);

static pure int indexOf(map<int, int> O, int v) =
    getFromMap(O, v);
```

History

The call stack is modelled using a sequence and three operations, 'addRecent', 'init' and 'last'. 'init' returns the prefix that is one less in length, and 'last' returns the element at the last index. 'addRecent' appends v to the call stack.

Listing 3.7: History functions

```
static pure seq<int> addRecent(int N, seq<int> H, int v) =
    H ++ v;

static pure seq<int> init(seq<int> H) =
    H[0..|H|-1];

static pure int last(seq<int> H) =
    H[|H|-1];
```

Derived state

We define three derived state variables: the set of all vertices V, the set of unvisited vertices $Unseen$, and the set of vertices that are visited but not explored $Live$. They are used only inside specification code. Their PVL definition are listed in Listing 3.8.

Listing 3.8: Derived state variables

```
static pure set<int> V(int N) = VHelper(N, 0);
static pure set<int> VHelper(int N, int i) =
    i < N ? {i} + VHelper(N, i+1) : set<int> {};

inline pure set<int> Unseen() = V(N) - Visited;

inline pure set<int> Live() = Visited - Explored;
```

3.4 Formalising the proofs

In this section we address how the observations mentioned in Subsection 3 can be formalised and translated into PVL. First we provide formal definitions for correctness criteria, and then we will prove that these criteria are met.

3.4.1 Predicates

We define predicates that eventually lead us to a formalised definition of an SCC. To reiterate, an SCC is a set of vertices that all have a path to each other, and the set is also maximal (no other vertex can be added). We define a *path* as a sequence of vertices that are connected by edges in the graph. In Listing 3.9 at line 1 and 2 we define the parameters to the predicate: G is the graph, x is the first vertex in the path, y is the last vertex in the path, and P stands for the path itself. Line 5 defines that all values in the path should be in range $[0 \dots N)$, and line 6 states that all vertices in the path (except the last one) have an edge to the next vertex.

Listing 3.9: The 'Path' predicate

```
1 static pure boolean Path(int N, seq<seq<boolean>> G,
2     int x, int y, seq<int> P) =
3     0 <= x && x < N && 0 <= y && y < N &&
4     0 < |P| && P[0] == x && P[|P| - 1] == y &&
5     (\forall int j; 0 <= j && j < |P|; 0 <= P[j] && P[j] < N) &&
6     (\forall int j; 0 <= j && j < |P| - 1; G[P[j]][P[j + 1]]);
```

Next, we introduce a *fitting path*. That is a *path* that is contained in some set C. We will use this indirectly in the formal definition for FSCC.

Listing 3.10: The 'FittingPath' predicate

```
1 static pure boolean FittingPath(int N, seq<seq<boolean>> G,
2     int x, int y, seq<int> P, set<int> C) =
3     Path(N, G, x, y, P) &&
4     (\forall int v; v in P; v in C);
```

Existentially quantifying over the path variable gives us 'ExFittingPath':

Listing 3.11: The 'ExFittingPath' predicate

```
1 static pure boolean ExFittingPath(int N, seq<seq<boolean>> G,
2     int x, int y, int len, set<int> C) =
3     (\exists seq<int> P; len <= |P|; FittingPath(N, G, x, y, P, C));
```

Then, we obtain the definition for FSCC listed in Listing 3.12. An FSCC is a set of vertices (C) where for all pairs of vertices (x and y) in C, there exist at least one path from x to y (line 3) and from y to x (line 4). All paths involved have all their elements contained in C.

Listing 3.12: The FSCC predicate

```
1 static pure boolean FSCC(int N, seq<seq<boolean>> G, set<int> C) =
2     (\forall int x; x in C; (\forall int y; y in C;
3         ExFittingPath(N, G, x, y, 1, C) &&
4         ExFittingPath(N, G, y, x, 1, C) ));
```

We note that singleton partitions are always an FSCC, since path of length 1 satisfy the ExFittingPath predicate, thus there is no need to specify that $x \neq y$. To obtain the SCC definition, we 'simply' add one more requirement: C is maximal. We encode this in Listing 3.13 by stating that there is no other FSCC Cp (line 3) which overlaps with C (line 4).

Listing 3.13: The SCC predicate

```
1 static pure boolean SCC(int N, seq<seq<boolean>> G, set<int> C) =
2     FSCC(N, G, C) &&
3     (\forall set<int> Cp; |Cp| > 0 && FSCC(N, G, Cp) && Cp != C;
4     !(\forall int x; x in C; (x in Cp)));
```

Since this property is not verified in this thesis, future work may consider alternative formulations if they are found to be better suited for verification, e.g.:

Listing 3.14: Alternative SCC definition

```
static pure boolean SCC(int N, seq<seq<boolean>> G, set<int> C,
    set<int> Visited) = FSCC(N, G, C) &&
    (\forall int x; x in Visited; (\exists int y; y in C;
    (ExPath(N, G, x, y, 1) && ExPath(N, G, y, x, 1)) ==> (x in C)));
```

Stated informally: C is an FSCC, and all visited vertices x that have a path from and to any node y in C are also contained in C. In other words: there cannot be any path from and to C where one of the vertices in the path (x) is outside C. When all reachable nodes are visited, this implies maximality of C. The reason I believe this formulation is easier to verify, is that it is an invariant that stays true throughout execution of the algorithm. It includes the set of $Visited$ vertices in the definition, meaning unreachable states are modelled explicitly. Hence it is therefore easier to use inside a universally quantified predicate which quantifies over *all* partitions (also the unreachable ones).

3.4.2 Algorithm invariants

In this subsection we will be going over the formalisation of invariants, and how they lead to a proof that guarantees that partitions in S are FSCCs. To keep verification times managable the algorithm is split up into four routines: SetBased, MarkVisited, StackCollapse and MarkExplored. SetBased corresponds to lines 1-25 in Listing 3.1, MarkVisited corresponds to lines 2-5, StackCollapse corresponds to lines 11-14, and MarkExplored to lines 20-22. Then we verify a set of properties for each of the subroutines.

Listing 3.15: MarkVisited routine

```
void MarkVisited(int v) {
    Visited = Visited + {v};
    O = append(O, v);
    R = push(N, R, v);
    H = addRecent(N, H, v);
}
```

Listing 3.16: SetBased routine

```
void SetBased(int v) {
    MarkVisited(v);

    int w = 0;
    while (w < N) {
        if (!(w in Explored)) {
            if (!(w in Visited)) {
                SetBased(w);
            } else {
                StackCollapse(v, w);
            }
        }
        w++;
    }

    if (v == top(N, R)) {
        MarkExplored(v);
    }

    H = init(H);
}
```

Listing 3.17: MarkExplored routine

```
void MarkExplored(int v) {
    Explored = Explored + part(N, S, v);
    tuple<seq<int>, int> t = pop(N, R);
    R = getFst(t);
}
```

Listing 3.18: StackCollapse routine

```
void StackCollapse(int v) {
    while (rep(N, S, v) != rep(N, S, w)) {
        tuple<seq<int>, int> t = pop(N, R);
        int r = getSnd(t);
        R = getFst(t);
        int newRep = top(N, R);
        S = uniteRoots(N, S, newRep, r);
    }
}
```

Two minor adjustments were made to the StackCollapse routine compared to Listing 3.1.

- The while condition has changed from `part(N, S, v) != part(N, S, w)` to `rep(N, S, v) != rep(N, S, w)`. This substitution is justified since partitions are only ever equal when their representatives are equal (by definition of the `part` function).

- The call to `unite` has been replaced by `uniteRoots`. These two methods share the exact same implementation body - the only change is in the contract: `uniteRoots` requires that the provided arguments are their own representatives in S, while `unite` does not. `uniteRoots` can then make one extra guarantee, namely that '`\result == S[r -> newRep]`'.

These two slight modifications improved the verification time significantly.

3.4.3 Basic invariants

First we verify observation 3: All roots in R are representative in S, and all vertices in the *Live* set have their representatives in R. Hollander [11] formulates this more strictly: every node v in *Live* has a unique representative in R. Mathematically he writes

$$\biguplus_{r \in R} S(r) = \mathit{Live}$$

and

$$\{S(v) \cap R \,|\, v \in \mathit{Live}\} = \{\{r\} \,|\, r \in R\}$$

in Section 4.3. We prove this property by first proving that it holds when the algorithm starts, then we prove that SetBased maintains the property. For this, we are also obligated to prove that MarkVisited, MarkExplored and StackCollapse all maintain the property as well.

In PVL we formulate the invariant as follows:

Listing 3.19: Invariants for R, *Live* and *Unseen*

```
1 (\forall int x; x in R; x == rep(N, S, x));
2 (\forall int x; x in Live(); rep(N, S, x) in R);
3 (\forall int x; x in Unseen(); part(N, S, x) == {x});
```

Proof of establishment: Initially both R and *Live* are empty, automatically proving the invariant 1 and 2. Statement 3 is proven by contradiction. Every vertex is initially in *Unseen* so we assume that for an arbitrary x that $S(x) \neq \{x\}$. Since initially every vertex x is its own representative we have $x \in S(x)$. That means there must exist some element y where $y \neq x \wedge y \in S(x)$. But that means that $rep(N, S, y) = x$ which contradicts our precondition that states $rep(N, S, y) = y$.

Now we present the proof of preservation of these three properties for each of the algorithm routines:

- For MarkVisited: v is added to *Visited*, but *Explored* remains unchanged, thus v effectively v is added to *Live*. And since v was in *Unseen* before the call, v is its own representative, thus the representative of v is added to R, preserving the invariants on line 1 and 2. The *Unseen* set just shrinks, but no partition changes, so the invariant on line 3 still holds, albeit with one element less in *Unseen*.

- For StackCollapse: in the loop the partitions of two roots get merged. The *Unseen* and *Live* sets do not change. This proves statement 3. R decreases by one element, but all the remaining roots are still their own representatives (the partition of the top root has just grown, but the representative did not change). This proves statement 1. Statement 2 is proven by the fact that all element in *Live* that were represented by r are now represented by *newRep*, which is the new top element of R. All other *Live* elements keep their original representatives.

- For MarkExplored: Statement 1 is proven following the same reason as for StackCollapse. In MarkExplored the *Unseen* set shrinks because *Explored* grows by $S(v)$, but all remaining *Unseen* vertices are still in their own singleton partition, proving statement 3. Statement 2 is proven by the fact that all remaining elements in *Live* are not represented by v, and thus their representatives are still in R.

- For SetBased: All three properties are maintained by every subroutine (including the recursive call), thus all three properties can be maintained as a loop invariant and added as preconditions and postconditions.

All of the proofs discussed until this point were already formalised and verified by Hollander, but two gaps were left open in that formalisation. These gaps are:

- Hollander's formalisation contains an assumption in the body of the loop in StackCollapse: `assume v in Live();`. If v is really in the *Live* set at that point, then we should be able to write it as an assertion and verify that it holds.

- Hollander does not verify the most important postcondition of SetBased, namely that all partitions in S that were found are in fact SCCs in G.

The next two paragraphs aim to fill these gaps, however we also do not succeed completely. Firstly, we verify the assumption in the loop body of StackCollapse stating that $v \in Live$. Secondly, we verify that all partitions in S are FSCCs throughout execution of the algorithm. The maximality proof is not formalised and not implemented in our PVL code.

3.4.4 Proof of 'v in Live'

We continue by following Hollander's recommendation which is proving that *Live* is monotonic. We prove that, from the perspective of a single call to SetBased, it can only grow. From there we can conclude that, once v is added to *Live* by MarkVisited, it stays in the *Live* set throughout the entire successor loop. We obtain the following postcondition for SetBased regarding *Live*:

Listing 3.20: Postcondition for SetBased regarding *Live*

```
ensures \old(Live()) <= Live();
void SetBased(int v) { /*implementation*/ }
```

The proof for this postcondition is as follows: MarkVisited adds v to *Live*, we denote this as $Live_0 = \backslash old(Live) \cup \{v\}$. Trivially, we have $\backslash old(Live) \subset Live_0$. Then we prove $Live_0 \subseteq Live$ as the loop invariant for the outer while-loop at lines 5-14 in Listing 3.16. The proof goes by induction: initially $Live_0 = Live$ which proves the base case. For the induction step we denote '*Live* as observed after the ith iteration' as $Live_i$. Our induction hypothesis is $Live_0 \subseteq Live_i$. Since the recursive call at line 8 is the only place where *Live* gets updated in the loop, and the contract of SetBased guarantees monotonicity already, we can conclude that $\forall_{k \in [1..\#iterations]} Live_{k-1} \subseteq Live_k$. Using the induction hypothesis we obtain $\forall_{k \in [1..\#iterations]} Live_0 \subseteq Live_{k-1} \wedge Live_{k-1} \subseteq Live_k$, thus $Live_0 \subseteq Live_k$. Thus after the loop we can conclude $Live_0 \subseteq Live$. Now our only remaining proof obligation is to prove that MarkExplored also guarantees that $\backslash old(Live) \subseteq Live$. Since MarkExplored grows the *Explored* set, we must therefore prove that at line 2 in Listing 3.17 $S(v)$ and $\backslash old(Live)$ are disjoint.

Proof of '$\backslash old(Live)$ and $S(v)$ are disjoint'

This proof works as follows: We maintain an invariant stating that (1) every vertex encountered after v cannot be in $\backslash old(Live)$ and (2) every vertex in $S(v)$ is encountered after v (or it is v itself). Since $v \notin \backslash old(Live)$, we obtain that $S(v)$ can't overlap with $\backslash old(Live)$.

We begin by defining three extra invariants, *Roots are ordered*, *Partitions are ordered* and *old Visited Before v*:

Definition 4. *Roots are ordered: every root in R is encountered $Before$ its successor root, one higher up in the stack.*

Definition 5. *Partitions are ordered: every representative in S is encountered $Before$ all other vertices in its partition.*

Definition 6. *old Visited Before v: every vertex that was already $Visited$ before v is encountered $Before$ v.*

PartitionsAreOrdered will eventually be used in the body of MarkExplored to prove disjointness of $\backslash old(Live)$ and $S(v)$, and *RootsAreOrdered* is necessary to prove preservation of *PartitionsAreOrdered* in StackCollapse.

In PVL we implement:

Listing 3.21: Invariant definitions vor '$v \in Live$' proof

```
static pure boolean RootsAreOrdered(seq<int> R, map<int, int> O) =
    (\forall int i; 0 <= i && i < (|R| - 1); Before(O, R[i], R[i+1]));

inline static pure boolean PartitionsAreOrdered(int N, seq<int> S,
    set<int> Visited, map<int, int> O) =
    (\forall int x; x in Visited; Before(O, {:rep(N, S, x):}, x));

static pure boolean Before(map<int, int> O, int x, int y) =
    indexOf(O, x) <= indexOf(O, y);

inline static pure boolean OldVisitedBeforeV(map<int, int> O,
    set<int> oldVisited, int v) =
    (\forall int x; x in oldVisited; indexOf(O, x) < indexOf(O, v))
```

We make the following three observations:

- The predicates are partially defined - they are only defined for elements which are already encountered (and thus present in the key set of the map).
- $Before$ is both reflexive and transitive, i.e. $Before(O, x, x)$ is *true* for any visited x, and $(Before(O, x, y) \wedge Before(O, y, z)) \rightarrow Before(O, x, z)$.

We now prove *RootsAreOrdered* (Definition 4) for every subroutine of the algorithm:

- At the start of the algorithm R is empty, so the universal quantifier evaluates to *true*.
- For MarkVisited, *RootsAreOrdered* is trivially maintained, since $indexOf(v)$ equals $|Visited|$ which is more than the index of the previous root because all other elements in O have an index lower than $|Visited|$. Unless R was empty, we know that R_{top} is present in O because R is a subset of $Visited$.

- For StackCollapse, the root stack gets popped in a loop, so every iteration there is one root less, but the order still stays preserved since R stays a prefix of $\backslash old(R)$.
- For MarkExplored: Idem.
- For SetBased the invariant holds trivially since it is maintained by all subroutines.

Proof of *PartitionsAreOrdered* (Definition 5) for every subroutine of the algorithm:

- Initially S contains only singleton partitions, so *PartitionsAreOrdered* is established by $Before$ being reflexive.
- MarkVisited and MarkExplored does not change S, so *PartitionsAreOrdered* is trivially preserved from the precondition.
- StackCollapse preserves *PartitionsAreOrdered* since in the loop the new representative of the merged partition is the 'oldest' root (of the two). Let us denote the state of S after the ith iteration as S_i. We then obtain $S_i(newRep) = S_{i-1}(newRep) \cup S_{i-1}(r)$. To prove $\forall x \in S_i(newRep).\, Before(O, newRep, x)$ we make a case distinction:
  1. $x \in S_{i-1}(newRep)$: We already know that $Before(O, newRep, x)$ holds from *PartitionsAreOrdered* from the loop invariant.
  2. $x \in S_{i-1}(r)$: We know that $Before(O, r, x)$ from *PartitionsAreOrdered* from the loop invariant. From *RootsAreOrdered* we know that $Before(O, newRep, r)$, thus applying transitivity yields $Before(O, newRep, x)$.
- SetBased preserves the invariant as well since MarkVisited, MarkExplored, StackCollapse and the recursive call preserve it. Like *RootsAreOrdered*, *PartitionsAreOrdered* can simply be added as a loop invariant.

Trivially $OldVisitedBeforeV(O, \backslash old(Visited), v)$ also holds in the outer loop of SetBased, since it can be maintained as a loop invariant.

Now that *PartitionsAreOrdered* is proven, we use it to prove disjointness of *oldLive* and $S(v)$. Namely, by $OldVisitedBeforeV$, we obtain $\forall x \in oldLive.\, indexOf(O, x) < indexOf(O, v)$, and $\forall x \in S(v).\, indexOf(O, v) \leq indexOf(O, x)$ by *PartitionsAreOrdered*. Thus v acts as a pivot element in O. Because O contains only unique elements, we conclude that there is no overlap between *oldLive* and $S(v)$, thus $\backslash old(Live) \subseteq Live$ stays preserved after line 2 in MarkExplored $\square$.

3.4.5 Proof of 'FSCC'

In this section we prove that every partitions remains strongly connected throughout execution of the algorithm. We begin with the following insight: If there is a vertex that can reach every other vertex in a partition, and this vertex can also be reached from every other vertex in the partition, then, by path concatenation, every vertex can reach every other vertex in said partition. Then, we use the representative of a partition as the connection point for every path concatenation. We can extend this concept to *Fitting Paths* (Listing 3.10) as well. We will introduce the following shorthand notation for fitting paths: $ExFittingPath(N, G, a, b, 1, C) \equiv a \xrightarrow[C]{}^* b$. In PVL we prove that the following statement holds:

$$x \xrightarrow[C_1]{}^* y \wedge y \xrightarrow[C_2]{}^* z \wedge (C_1 \cup C_2) \subseteq C_3 \implies x \xrightarrow[C_3]{}^* z$$

The proof can be found in Appendix B. We use *fitting* paths (instead of regular paths) since those are used by the definition of an FSCC (see Definition 2).

We will introduce another piece of notation: $rep(N, S, x) \equiv S^N[x]$. Then, we prove that $\forall_{x\in[0\ldots N)}\, x \xrightarrow[S(x)]{}^* S^N[x] \wedge S^N[x] \xrightarrow[S(x)]{}^* x$ holds at every point in the algorithm. In PVL we implement:

Listing 3.22: 'ConnectedPartitions' definition

```
static pure boolean ConnectedPartitions(int N, seq<seq<boolean>> G,
    seq<int> S) =
    (\forall int x; 0 <= x && x < N; CP(N, G, S, x));

static pure boolean CP(int N, seq<seq<boolean>> G,
    seq<int> S, int x) =
    ExPathToRep(N, G, S, x) && ExPathFromRep(N, G, S, x);

static pure boolean ExPathToRep(int N, seq<seq<boolean>> G, seq<int> S,
    int x) =
    ExFittingPath(N, G, x, rep(N, S, x), 1, part(N, S, x));

static pure boolean ExPathFromRep(int N, seq<seq<boolean>> G,
    seq<int> S, int x) =
    ExFittingPath(N, G, rep(N, S, x), x, 1, part(N, S, x));
```

- *ConnectedPartitions* states that *CP* holds for all vertices.
- *CP* states that there exists a path from x to its representative and there exists a path from x's representative to x.

Once we prove $ConnectedPartitions(N, G, S)$, we can then claim that every partition is an FSCC (by path concatenation, explained earlier). The proof follows the same structure as earlier proofs: We prove that the condition holds at the start of the algorithm, and then prove preservation for the SetBased routine.

To prove the *ConnectedPartitions* invariant, we define two more invariants first:

Definition 7. *The root path is the path that the algorithm finds by traversing the graph depth-first. Informally: There exists a path from every root to the 'next' root in R. These paths are contained within the set that is the partition of the first root unioned with the second root.*

$$RootPath(N, G, R, S) \triangleq \forall_{i\in[0\ldots|R|-1)}\, R[i] \xrightarrow[S(R[i])\,\cup\,\{R[i+1]\}]{}^* R[i+1]$$

Definition 8. *$HistoryRepresentedByRoots$: R is a subsequence of H and all elements in H have their representatives in R, in the same order.*

From this definition of $HistoryRepresentedByRoots$ it follows that $last(H) = v$ and $S^N[v] = R_{top}$. Together, these two definitions assist in proving that for every partition and for every vertex in the partition there exists paths from and to the representative.

In PVL we write:

Listing 3.23: ‘RootPath’ definition

```
static pure boolean RootPath(int N, seq<seq<boolean>> G, seq<int> R,
    seq<int> S) =
    (\forall int i; 0 <= i && i < (|R|-1);
        (\let set<int> C = part(N, S, R[i]) + {R[i+1]};
            ExFittingPath(N, G, R[i], R[i+1], 1, C)));
```

Listing 3.24: ‘HistoryRepresentedByRoots’ definition

```
static pure boolean HistoryRepresentedByRoots(int N, seq<int> H,
    seq<int> R, seq<int> S) =
    (|H| == 0 && |R| == 0)
    ||
    (|H| > 0 && |R| > 0 &&
        HistoryRepresentedByRoots_NonEmpty(N, H, R, S));

static pure boolean HistoryRepresentedByRoots_NonEmpty(int N,
    seq<int> H, seq<int> R, seq<int> S) =
    (\let int v = last(H); rep(N, S, v) == top(N, R) && (
        v == rep(N, S, v)
            ? ((|R| >= 2 && HistoryRepresentedByRoots_NonEmpty(N,
            init(H), getFst(pop(N, R)), S)) || (H == [v] && R == [v]))
            : (|H| > |R| && (\let seq<int> initH = init(H);
                rep(N, S, v) == rep(N, S, last(initH)) &&
                HistoryRepresentedByRoots_NonEmpty(N, initH, R, S))))
    );
```

Whilst the PVL encoding of ‘RootPath’ is a rather direct translation of the mathematical definition, the encoding of ‘HistoryRepresentedByRoots’ is not. It is best explained using a picture, shown in Figure 3.5. It is an inductive definition, starting at the last elements of H and R. It states that the currently considered last element of H is equal to R_{top}, and then all preceding element of H are represented by the preceding elements of R. If not, then the last two elements of H have the same representative, which is still R_{top}. The function is called recursively with $init(H)$ and R to check that the preceding elements of H are represented by R. The recursion stops when $H = R = [v_0]$. A special case is defined for $H = [] \wedge R = []$.

Proof of *RootPath*

- When the algorithm starts $|R| = 0$, so there are no roots that can have a path. The universal quantifier evaluates to *true*.
- MarkVisited preserves the invariant since we push v on top of R. There is an edge from v's predecessor u: $u \rightarrow v$, and since by *ConnectedPartitions* we know that $S^N[u] \xrightarrow[S(u)]{}^* u$, we concatenate the edge $u \rightarrow v$ to this path to obtain $S^N[u] \xrightarrow[S(u) \cup \{v\}]{}^* v$. From *HistoryRepresentedByRoots* (Definition 8) we know that before MarkVisited we had $S^N[u] = R_{top}$, so we can substitute $S^N[u]$ and $S(u)$ for $S^N[\backslash old(R_{top})]$ and $S(\backslash old(R_{top}))$. For all the preceding roots we re-use the invariant from the precondition, which means we can now

unify: $\forall_{i \in [0 \dots |R|-2)} R[i] \xrightarrow[S(R[i]) \cup \{R[i+1]\}]{}^{*} R[i+1] \wedge R[|R|-2] \xrightarrow[S(R[|R|-2]) \cup \{R_{top}\}]{}^{*} R_{top} \implies$ $\forall_{i \in [0 \dots |R|-1)} R[i] \xrightarrow[S(R[i]) \cup \{R[i+1]\}]{}^{*} R[i+1]$ which then re-establishes $RootPath(N, G, R, S)$.

- StackCollapse preserves the *RootPath* invariant trivially, since in every iteration the top element of R is popped, all previously established paths of the predecessors still exist. Only the partition of the top root is changed, but this partition is not involved in any of the existing fitting paths of all previous roots (just the top root itself is).
- MarkExplored preserves *RootPath* for the same reason that StackCollapse preserves it.
- SetBased trivially preserves $RootPath(N, G, R, S)$ since MarkVisited, StackCollapse and MarkExplored all preserve it.

We will now prove that $HistoryRepresentedByRoots(N, H, R, S)$ holds for all subroutines of the algorithm:

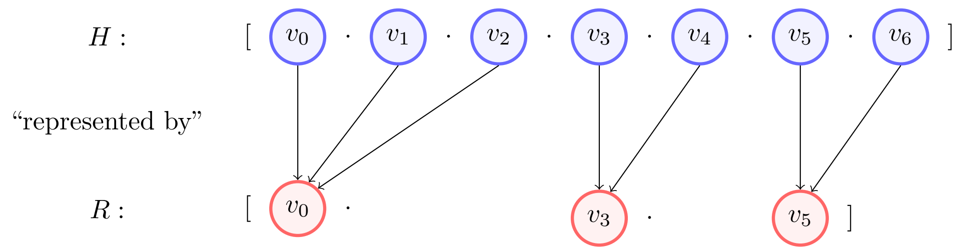


Figure 3.5: *HistoryRepresentedByRoots* example visualisation

Proof of ***HistoryRepresentedByRoots***

- In the beginning $|R| = 0 \wedge |H| = 0$, establishing the invariant trivially.
- MarkVisited: v is added to both R and H, preserving the invariant by folding the predicate once. We make the following case distinction:
  1. If $\backslash old(H) = [] \wedge \backslash old(R) = []$ then now $H = R = [v]$. In this case $v = v_0$. Trivially, we establish $HistoryRepresentedByRoots\_NonEmpty(N, [v], [v], S)$.
  2. Otherwise, we establish $HistoryRepresentedByRoots\_NonEmpty(N, \backslash old(H) \cdot v, \backslash old(R) \cdot v, S)$ using $|R| \geq 2 \wedge v = S^N[v] = R_{top} = last(H) \wedge$ $HistoryRepresentedByRoots\_NonEmpty(N, old(H), \backslash old(R), S)$.
- SetBased: the statement is added as a loop invariant because the invariant is preserved by both StackCollapse and by the recursive call. After the loop at line 14 in Listing 3.16 we obtain another case distinction - either (1) v was its own representative (and thus popped from R), or (2) it was not. Although in both cases the proof obligation is $HistoryRepresentedByRoots(N, init(H), R, S)$ the proofs are different.
  1. MarkExplored pops v from R, allowing us the unfold the definition of *HistoryRepresentedByRoots_NonEmpty* once, and re-establishing the invariant from the recursive call (1st branch in if-then-else at line 11).

2. v is not its own representative, so v now has the same representative as its predecessor. Hence we unfold the definition of $HistoryRepresententedByRoots\_NonEmpty$ once again, but we use the second branch of the if-then-else expression to re-establish the invariant (line 14, Listing 3.24). Note that we gain *HistoryRepresentedByRoots_NonEmpty(N, init(H), R, S)* as knowledge, which will correspond to *HistoryRepresentedByRoots_NonEmpty(N, H, R, S)* when the method returns (line 20, Listing 3.16).

- StackCollapse continuously pops from R, but it also makes all vertices that were represented by the old R_{top} now represented by the new R_{top}. The relation between H and R thus stays preserved. Figure 3.6 shows the relationship between H and R of the example (figure 3.5) after one call to `uniteRoots`. The PVL proof can be found in Appendix C.

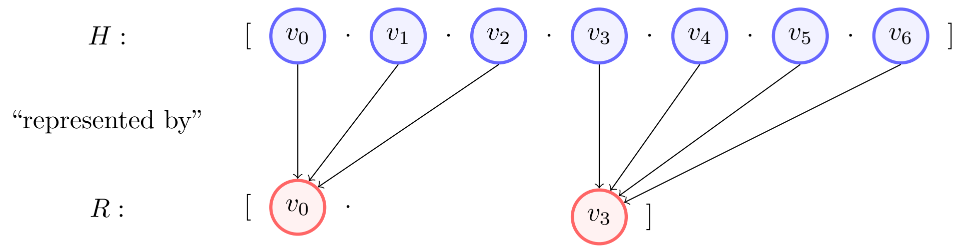


Figure 3.6: $HistoryRepresentedByRoots$ example, after one 'unite'

- MarkExplored: v is the top element of both H and R, so we can establish $HistoryRepresentedByRoots\_NonEmpty(N, init(H), getFst(pop(N, R)), S)$ afterwards (by unfolding line 13), or $HistoryRepresentedByRoots(N, [], [], S)$ in case H and R are now empty.

Now that $RootPath$ and $HistoryRepresentedByRoots$ (and especially $S^N[v] = R_{top}$) have been shown to hold throughout the entire algorithm, we can tie them together and prove $ConnectedPartitions$ (Listing 3.22).

Proof of $ConnectedPartitions$

- At the beginning, the invariant is proven by every vertex being its own representative in its own singleton partition. Since we define *path*s as sequences of vertices, the fitting paths in question are just paths of length 1.

- MarkVisited and MarkExplored: S is unchanged, so the invariant is proven from the precondition.

- StackCollapse: $ConnectedPartitions$ is broken during the loop, but re-established again afterwards (line 8. We split the proof for all $x. 0 \leq x < N$ in two cases: $S^N[x] \xrightarrow[S(x)]{}^* x$ and $x \xrightarrow[S(x)]{}^* S^N[x]$.

  1. 'From': ($S^N[x] \xrightarrow[S(x)]{}^* x$) is maintained by the loop: If during the while-body in StackCollapse x gets a new representative, then the fitting path from that representative to x can be established by concatenating $R[|R|-2] \xrightarrow[S(R[|R|-2]) \cup \{R_{top}\}]{}^* R_{top}$ (by $RootPath$)

and $R_{top} \xrightarrow[S(R_{top})]{}^* x$ (by $ExPathFromRep$), which obtains: $R[|R|-2] \xrightarrow[S(R[|R|-2])]{}^* x$.
After the stack is popped, we can substitute:
$R_{top} \xrightarrow[S(R_{top})]{}^* x$. Since $R_{top} = S^N[x]$ by ($HistoryRepresentedByRoots$) we can substitute again: $S^N[x] \xrightarrow[S(S^N[x])]{}^* x$ and again: $S^N[x] \xrightarrow[S(x)]{}^* x$.

Otherwise, if during the while-loop x does not get a new representative, then the pre-existing path $S^N[x] \xrightarrow[S(x)]{}^* x$ stays of course maintained.

2. 'To': ($x \xrightarrow[S(x)]{}^* S^N[x]$) is not maintained by the loop. Instead, for x that are members of the united partition, we prove that this path exists again after loop by performing 4 path concatenations and 2 substitutions. We know that v and w now share the same representative, which is R_{top}. In the following table we list the sub-paths and the reasons why we can prove they exist:

Path:	**Proof for its existence:**
$x \xrightarrow[\backslash old(S(x))]{}^* \backslash old(S^N[x])$	$ConnectedPartitions$ precondition (specifically: $ExPathToRep$).
$\backslash old(S^N[x]) \xrightarrow[S(R_{top})]{}^* \backslash old(S^N[v])$	$RootPath$ precondition: since every root has a fitting path to the next root, by concatenation every root as a path to the top root (which fits in the union of all their individual partitions).
$\backslash old(S^N[v]) \xrightarrow[\backslash old(S(v))]{}^* v$	$ConnectedPartitions$ precondition (specifically: $ExPathFromRep$).
$v \xrightarrow[\{v,w\}]{}^* w$	$v \to w$ is a direct edge.
$w \xrightarrow[\backslash old(S(w))]{}^* \backslash old(S^N[w])$	$ConnectedPartitions$ precondition (specifically: $ExPathToRep$)

We prove that $S(R_{top})$ subsumes all the sets containing these paths by definition:

$$S(R_{top}) = \bigcup_{indexOf(\backslash old(R), \backslash old(S^N[w])) <= i < |\backslash old(R)|} \backslash old(S(R[i]))$$

We can now prove that the concatenation of these 5 paths is also completely contained in $S(R_{top})$ and end up with $x \xrightarrow[S(R_{top})]{}^* \backslash old(S^N[w])$. Since the representative of w is not changed during the loop we know $\backslash old(S^N[w]) = S^N[w] = S^N[v] = S^N[x] = R_{top}$. Substituting gives us $x \xrightarrow[S(x)]{}^* S^N[x]$.

Vertices x that are not a member of the united partition simply retain their paths to their representatives.

Now that we have established the paths between x and $S^N[x]$ in both directions for all x. $0 \leq x < N$, we can re-establish $CP(N, G, S, x)$ (Listing 3.22, line 5). Thus $ConnectedPartitions(N, G, S)$ is preserved by StackCollapse. The PVL proof follows the same idea, and is listed in Appendix D.

- SetBased: Proven from MarkVisited, MarkExplored, StackCollapse and the recursive call to SetBased preserving the invariant.

With $ConnectedPartitions(N, G, S)$ now proven for every subroutine, we can conclude that all partitions remain an FSCC throughout the algorithm $\square$.

4. Results

In this chapter we summarise which properties were formalised and verified using VerCors. We give an overview of important invariants and how they relate to each other. We also present the verification times of the finalised PVL programs.

4.1 Verified properties

We started with the work of Hollander as our basis, and verified that the algorithm finds FSCCs (Definition 2) [11]. To reach this goal, we defined two additional properties: $v \in Live$ and *ConnectedPartitions(N, G, S)*. Multiple extra invariants were defined that eventually build up to these two properties. Figure 4.1 shows how these properties build up together.

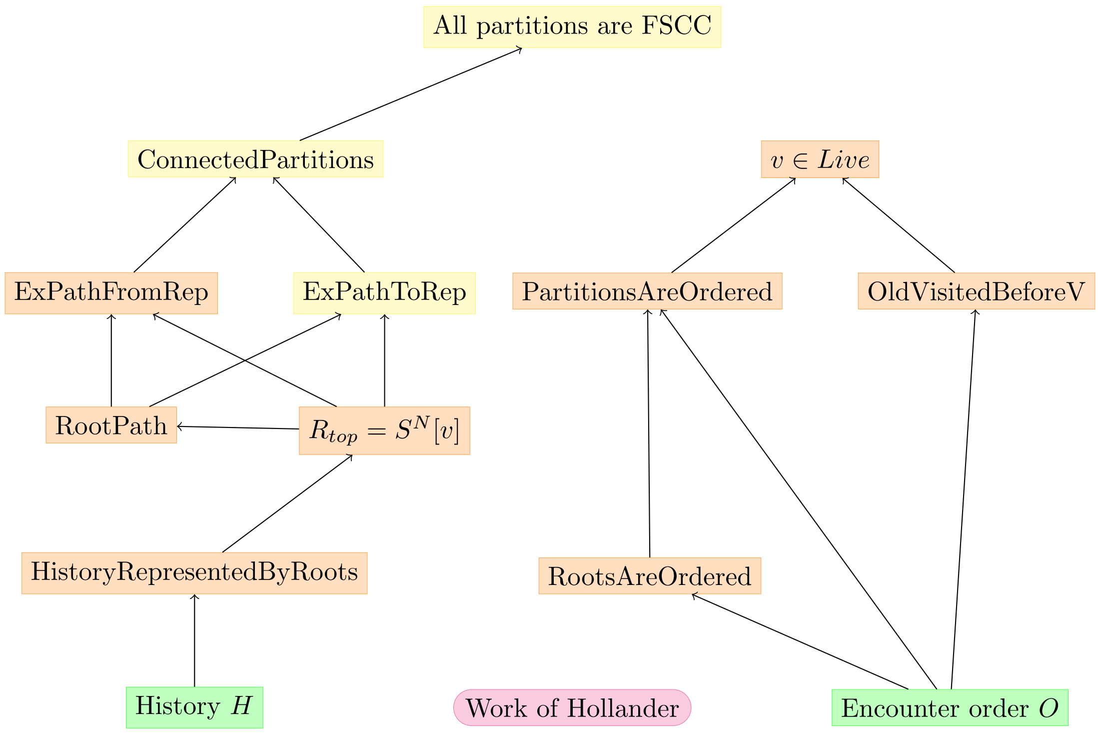


Figure 4.1: Dependency graph of properties and invariants

Invariants are highlighted in orange, data structures are highlighted in green, and other properties are highlighted in yellow. One can think of them as 'invariants-light' since they do hold before and after every subroutine, but not during the inner loop of the algorithm (in StackCollapse). Note that 'Work of Hollander' is an abstraction for a substantial amount of work. Hollander implemented the data structures Graph G, Stack R, Union-Find S and their operations (some of which required extra lemmas). He also verified basic invariants, and verified the invariants in Listing 3.19 partly. Lastly he defined PVL definitions for paths and the three types of SCCs. This node is essentially a dependency of all the other nodes in the figure, but we did not draw the corresponding arrows in order to not clutter the image. We also omitted our own lemmas from this figure.

4.2 Verification results

To prevent VerCors from hanging indefinitely, we apply techniques to limit the amount of information it has to reason with. One of these techniques is splitting up the algorithm over multiple subroutines, each with their own set of contracts, and each verified in their own file. These files are then verified in isolation. To ensure that the definitions used in these files remain in sync, we wrote them all in one big template file. A macro-expansion program then generates the files. We used this approach so that one change in the template file affects all 6 files simultaneously. We think this approach significantly helps with reproducing results, since it is not required to manually 'comment out' code that is not being verified. Consequently, there is zero chance that a human error occurs when manually checking whether two files use the same definition of a function. We will now list each generated file and explain its purpose:

- `Basic.pvl`: This file contains all the data structure operations and predicate definitions. It is verified separately so that we can use these definitions in the other files, without verifying the implementations again.
- `Main.pvl`: This file contains the main SetBased routine, and a lemma that proves that all found partitions are FSCCs. All properties mentioned in Section 4.1 are verified in this file, but just for the SetBased routine. The method bodies of MarkVisited, StackCollapse and MarkExplored are removed from this file, so that VerCors only needs to reason about using their contracts.
- `MarkVisited.pvl`: This file contains the full implementation of MarkVisited. In here all properties mentioned in Section 4.1 are verified for the MarkVisited routine.
- `MarkExplored.pvl`: This file contains the full implementation of MarkExplored. In here all of the mentioned correctness properties are verified for the MarkExplored routine.
- `StackCollapse__v_in_Live.pvl`: This file contains the full implementation of the program code of StackCollapse. This file verifies all properties from `Basic.pvl` as well as the invariant $v \in \mathit{Live}$ for the StackCollapse routine.
- `StackCollapse__FSCC.pvl`: This file contains the full implementation of the program code of StackCollapse. In here, all properties from `Basic.pvl` as well as the invariant *ConnectedPartitions* are verified for the StackCollapse routine.

4.2.1 Measurements

Table 4.1 shows the time that it took to verify each file [26], running on my Intel(R) Core(TM) i7-5500U CPU @ 2.40 GHz (4 CPUs) laptop with 12288MB of RAM running Windows 10 and Oracle JDK 17.0.2. 3 runs were conducted per file, and the table shows the average times.

File	**Average verification time (across 3 runs)**
`Basic.pvl`	34 s
`Main.pvl`	97 s
`MarkVisited.pvl`	41 s
`MarkExplored.pvl`	20 s
`StackCollapse__v_in_Live.pvl`	28 s
`StackCollapse__FSCC.pvl`	67 s*

Table 4.1: Verification times

Here we can see that the verification times are still somewhat acceptable for a fast iteration cycle, allowing future researchers to continue on this work. At the very least any consumer-grade computer released in the last 5 years should be able to verify these files within a reasonable time frame.

4.2.2 A note on StackCollapse

Note that StackCollapse has been split in two files, and the latter (`StackCollapse__FSCC.pvl`) takes 67 seconds to verify on average. This is a slightly misleading number since a handful of lemma calls in this file were disabled, while the postconditions were assumed at the call site. For all lemmas we can enable them and assert their postconditions and then verify the file again, one by one for each lemma. Enabling a lemma easily adds 10 to 20 seconds to the verification time. Unfortunately, we could not enable all lemmas at once since then VerCors would no longer verify the file - it would just run seemingly infinitely. One could argue that the same macro-expansion technique could be used to limit prover knowledge, but we opted against this given that these lemma definitions are only used in one place.

5. Related work

In this chapter we discuss other works in which formal methods were applied to verify SCC algorithms.

5.1 Bloemen's SCC algorithm in Isabelle by Vincent Trélat

In this work, Trélat et al. present a formalisation [28] of the sequential SCC algorithm described by Bloemen using the Isabelle [12] interactive theorem prover. There are three key differences in their encoding compared to ours.

1. They use a functional model where we use an imperative model - i.e. the main routine of the algorithm (which we call SetBased) is a pure function `dfs` which takes an environment and a vertex as arguments, and outputs a new environment as a result. The environment (often referred to as `e`) contains all the state variables such as the stack R, union-find S and $Visited$ and $Explored$ sets.
2. They 'implement' the union-find S using a function of type *'v → 'v set*, i.e. a function that returns the partition for a given vertex. This is more abstract than the sequence representation that we inherit from Hollander.
3. A set of found SCCs is accumulated in the environment whilst we do not use an extra variable in our formalisation - we simply make claims about the existing union-find structure S.

With this encoding, Trélat is able to verify that all found partitions are indeed *maximal* SCCs. Interestingly, they do not encode the notion of *fitting*ness. One could argue this is not required because any *maximal* SCC is also $fitting$. The proof by contradiction is left as an exercise to the reader. The key invariant that leads them to this conclusion is the fact that every path from vertex $m \in R$ to $n \in R$ where m is higher in the stack than n has not been followed yet. This property is named `reachable_avoiding`. When the algorithm finished visiting all successors and v is its own representative, then that means a path to a representative lower in the stack cannot exist, hence v's partition is a *maximal* SCC.

At a glance it seems possible to use this approach as well in our own encoding since we can easily check that that m is higher in R then n by comparing $indexOf(O, m) > indexOf(O, n)$ and conjuncting that with the *RootsAreOrdered* invariant.

5.2 Gabow's SCC algorithm using refinement in Isabelle by Peter Lammich

In this work Lammich presents a formalisation [15] of Gabow's SCC algorithm [8]. Similar to Bloemen's algorithm it is also based on Munro's original algorithm, hence it makes for a good candidate for comparison. Gabow's SCC algorithm is different in the following ways:

- Instead of storing partitions using a union-find, this algorithm uses a custom data structure designed by Gabow. Partitions of nodes are called *cnodes*; I like to use the mnemonic *c*ollapsed *node*, or *c*ollection of *nodes*.
- Gabow's SCC algorithm does not keep track of a stack of roots (R), instead, the *cnodes* live directly on the stack. This stack is referred to as 'the path'.

Lammich defines this formalisation using Isabelle/HOL. He verifies that the algorithm computes maximal SCCs and that found *cnodes* are topologically ordered. He does this by first verifying the algorithm using abstract data structures, and then proving that Gabow's data structure refines the abstract data structures. Then, he proves one more refinement using efficient data structures such as arrays and hash tables, and from this encoding he generates Standard ML code.

Maximality of SCCs is proven by the fact that there are no more unvisited outgoing edges from the latest cnode when it is popped from the path. One invariant used is that the *Done* set remains closed under transition, i.e. all nodes in *Done* only have edges that lead to other vertices that are also in *Done*. Note that the *Done* set in Lammich's terminology is equivalent to *Explored* in Bloemen's terminology.

5.3 Model checking UFSCC using TLA$^+$ by Jaco van de Pol

In this work Van de Pol presents a case study [19] of using the TLC model checker to check correctness of the parallel UFSCC algorithm presented by Bloemen. He model checks the algorithm for a number of small example graphs (at most 4 vertices), and 2 execution threads. Through experimentation and slight modifications he attempts to gain a better understanding of the algorithm, and its invariants that lead to correctness. He shows that the algorithm indeed guarantees maximal SCCs for the example graphs. This of course does not demonstrate that the algorithm behaves according to specification for any arbitrary graph, hence it is very different from our approach, but as far as we are aware, this is the first attempt at formally proving correctness of the parallel UFSCC algorithm.

6. Conclusion

We successfully verified that Bloemen's sequential SCC algorithm partitions the graph into FSCCs, meaning the partitions are both *strongly connected* and *fitting*. To complete this task, we utilised nearly every feature that VerCors has to offer for verification of sequential programs. We answer *RQ1* using the following summary of techniques used: We used the axiomatic data types `set`, `seq` and `map` to implement the data structures used by the algorithm. We wrote custom lemmas to prove invariants that were otherwise too hard for VerCors to verify. These lemmas mainly revolved around proving proving preservation of invariants during loops. Especially the invariants *HistoryRepresentedByRoots* and *ExPathToRep* required many lemmas. Some of our first-order logic formulas contain triggers to make sure that the back-end prover instantiates universally quantified expressions using the correct identifier. We split the algorithm up in multiple subroutines, each verifying a defined part of the algorithm. As it turned out, the StackCollapse routine was especially difficult to verify since it required many more lemmas than the other subroutines. When they were all implemented VerCors could not verify the file anymore, so they were verified independently. When disabled, their postconditions would be assumed at the call site in order to let the verification complete successfully.

6.1 Reflection

Looking back at this project, there were several hurdles to overcome. Some of the learned lessons are listed below. This list is my answer to *RQ2*, although this is of course subjective.

1. To start off, to formally verify an algorithm, one needs to understand it at a sufficiently deep level. Invariants need to be identified, and formulated in first-order logic. Some invariants are simple, and are easily maintained by the fact that some data structures only grow. Such invariants can usually be verified by VerCors right away, e.g. 'v stays in the $Visited$ set.' Other invariants require more effort to verify, e.g. $ConnectedPartitions$. It was necessary to write extra lemmas to convince VerCors of the existence of certain paths. Especially difficult was proving the path $v \to^* rep(N, S, v)$ since that invariant is temporarily broken during the inner loop in StackCollapse. It has to be re-proven again after the loop, using a concatenation of 5 sub-paths, all of which also required extra loop invariants to prove that they exist. Some of these loop invariants also required extra lemmas to prove preservation, so it seemed like a never-ending task. One might say that $v \to^* rep(N, S, v)$ is not an invariant, since it does not hold 'all the time' and I think that is the main takeaway: properties that hold in more than one place, but do not hold 'all the time' are the most difficult to verify.

2. To prove the more difficult invariants, one needs to write extra lemmas that prove preservation of some invariant in some specific context. Such lemmas can be trivial, or they can call into other lemmas, or into themselves. This also touches upon another point: with VerCors all the proof techniques at your disposal are essentially 1. simple constructive proofs, 2. proofs by contradiction, and 3. proofs by induction. This means that other proof techniques, e.g. pigeon hole proofs can only be used when decomposed into multiple induction and contradiction proofs. For example: in StackCollapse to prove that $v \to w$ is actually a back-edge, we need to prove that a path $w \to^* v$ already exists. The pigeon-hole proof for this looks as follows:

   (a) We know that both $v \in Live$ and $w \in Live$

(b) From the invariant $\forall_{x \in Live}\, rep(N,S,x) \in R$ we know that both $rep(N,S,v) \in R$ and $rep(N,S,w) \in R$.

(c) We also know that $rep(N,S,v) = R_{top}$. Since $rep(N,S,v) \neq rep(N,S,w)$ it follows that $rep(N,S,w)$ is somewhere lower in the stack (pigeon hole principle).

(d) Therefore, by the *RootPath* invariant, we know that there exists a path $rep(N,S,w) \rightarrow^* rep(N,S,v)$ (because all roots have a path to the top element of R).

(e) From *ConnectedPartitions* we know that $w \rightarrow^* rep(N,S,w)$ and $rep(N,S,v) \rightarrow^* v$ exists.

(f) The old representatives of v and w stay a member of their partitions, hence we can prove the path $w \rightarrow^* v$ exists by concatenating $w \rightarrow^* \backslash old(rep(N,S,w)) \rightarrow^* \backslash old(rep(N,S,v)) \rightarrow^* v$.

While this manual proof is similar to the mechanised proof described in paragraph 3.4.5 (in fact, it formed the basis for it), it is also simpler in the sense that we did not have to provide an explicit witness from the path $rep(N,S,w) \rightarrow^* rep(N,S,v)$. We also implicitly assumed that this path stays within the partition when the partition is finally merged, but this is something that requires extra effort when formalised using a static verifier. One then needs to prove that all the sub-paths remain inside the partition while the loop is busy. The takeaway from this is to be aware of your implicit assumptions, and make them explicit when formalising the proof for a deductive verifier. Making this formalisation concrete is what causes formal proofs to take much more time than manual pen-and-paper proofs.

3. When formalising proofs in VerCors, it is required that the data structure operations are also verified, while in pen-and-paper proofs these are typically assumed to be correct. Thus formal verification requires extra effort for the verification of the contracts of these data structure operations. Luckily, I was able to thank Hollander for his effort in this area as he already implemented most of the data structure operations.

4. From our perspective, SMT solvers are essentially black boxes. They are sound, but in my experience Z3 has been acting very inconsistently. Sometimes being able to prove some a certain statement instantly, and sometimes hanging seemingly forever. SMT solvers are also notorious for struggling with existentially quantified formulas, hence it is usually required to specify an explicit witness. Often I had to rephrase the formulation entirely using witness values directly - the map-encoding of the encounter order is an example of this. During development, I used a PVL sequence before, whilst 'indexOf' performed a linear search. Its contract required that the to-be-found value is in the sequence ($x \in xs$), which is then translated by VerCors into an existential quantifier (the value 'exists' in the sequence). Verification time blew up because of this. Furthermore, SMT solvers also seem to struggle when their pool of knowledge grows 'big', not being able to do inferences anymore that used to work before. One trick for dealing with this is to factor out parts to separate methods such that they can be verified with just the knowledge that is strictly required for them, however this approach isn't always viable. Other tricks that I have used are: (1) using explicit assert statement to unfold definitions or draw preliminary conclusions and (2) use triggers to make sure that the proper instantiation of a universally quantified formula gets added to the pool of knowledge.

5. To stabilise verification times I have customised the build of VerCors, such that it uses only 1 parallel Silicon verifier. One other customisation was to disable trigger generation for quantified expression (recommended by Hollander). In this case study, the generated triggers from

VerCors performed worse than the triggers inferred by Silicon/Z3. This feature has also been removed from VerCors since version 2.0.0, which means that others won't be running into this problem again.

6. To prove universally quantified formulas, stating some fact about elements in a range, or elements in a sequence, lemmas containing loops can be used. This essentially corresponds to strong induction. To prove $\forall x \in [0..N).P(x)$ one writes the loop invariant `(\forall int x; 0 <= x && x < i; P(x))` where i is the loop iterator variable that loops from 0 to N (inclusive). While to some this is natural, to me it was not always obvious that this construct should be used in certain places. Until halfway through the project, I was very much relying on the automatic inference capabilities of Silicon/Z3, rather than to write lemmas with such explicit loop invariants.

6.1.1 Recommendations

To conclude, I would like to give the following two recommendations to other researchers who use VerCors:

- Do not try to verify properties that are not invariants right away. Instead, begin with verifying the obvious invariants, and try to find invariants for properties that can be used inside larger proofs for the correctness properties.
- Do not hesitate to create auxiliary variables. They can serve as explicit witnesses for the to-be-proven invariants. This can in turn then help the underlying solver.
- When stuck at the verification of some property, make a proof on paper of why it should hold. Then encode all the proof steps as explicit assertions in the program. Chances are, one of the assertions can't be verified, because some knowledge is missing for the prover. Then, to solve this, add extra assertions, invariants or lemmas (depending on the context).
- When dealing with long verification times, try to keep Z3's pool of knowledge as small as possible. More knowledge lead to longer verification times because for Z3 it becomes harder to make the desired inferences, i.e. 'more knowledge' also means 'more pollution'. This applies especially to quantified expressions, since Z3 will have to find instantiations of these expressions that are then used later on for the next part of the proof. Triggers can also help here. Sometimes it can be better to formulate a property using an inductive definition that is unfolded manually, then to encode it using a universal quantifier.

7. Future work

7.1 Proving maximality

Proving that the algorithm decomposes the graph into *maximal* SCC is a non-trivial task. This property is not trivial invariant that stays true the entire time. Hence, one needs to find invariants and other properties that eventually lead up to the notion of maximality at line 20 in Listing 3.1. Paragraph 3.2.1 already sketched an intuition for the maximality proof, so one would need to further elaborate on this and formalise this. Listing 3.14 can serve as an inspiration for this. Additionally, one may want to verify that every reachable node is actually visited (and thus a member of the *Visited* set when the algorithm terminates). This property could be formalised as $v_0 \in Visited \wedge \forall x \in Visited.\ \forall y \in \mathbf{succ}(x).\ y \in Visited$. Additionally one may want to verify that, after the algorithms successor loop, the set of descendants of v (the successors of v, and their successors, and their successors, etc) consists of only vertices that are already *Explored*, or vertices that are in the same partition as v. Alternatively, future researchers may decide to attempt to use the solutions for proving maximality from Trélat [28] or Lammich [15] as both their solutions seem like they can be ported to VerCors/PVL.

If VerCors is used to verify the maximality proof, then it is highly advised to verify the invariants for maximality separately, and leave the already-proven properties as assumptions. This gives the best chance on the verifier producing an output, rather than (perceived) non-termination.

7.2 Going concurrent

To verify Bloemen's parallel $UFSCC$ algorithm [2], additional prerequisites need to be fulfilled. First, because of global coordination concerns, the union-find's 'unite' operation is no longer guaranteed to make the first argument the new representative of the merged partition. Instead, it is left out as an implementation detail. Bloemen's implementation uses the vertex with the highest hash code as the new representative of the merged partitions. The invariants *PartitionsAreOrdered*, *RootPath* and *HistoryRepresentedByRoots* need an extra level of indirection to account for this. Their PVL definitions would change to something much like Listing 7.1.

Listing 7.1: Updated RootPath and PartitionsAreOrdered definitions

```
//changed: ExFittingPath(N, G, R[i], R[i+1], 1, C) --->
//         ExFittingPath(N, G, rep(N, S, R[i]), rep(N, S, R[i+1]), 1, C)
static pure boolean RootPath(int N, seq<seq<boolean>> G, seq<int> R,
    seq<int> S) =
    (\forall int i; 0 <= i && i < (|R|-1);
        (\let set<int> C = part(N, S, R[i]) + {R[i+1]};
            ExFittingPath(N, G, rep(N, S, R[i]), rep(N, S, R[i+1]), 1, C)));

//changed: rep(N, S, x) --->
//         first(N, S, x)
static pure boolean PartitionsAreOrdered(int N, seq<int> S,
    set<int> Visited, map<int, int> O) =
    (\forall int x; x in Visited; Before(O, first(N, S, x), x));
```

In this example `first` is a made up function that returns the first encountered vertex in $S(x)$. The `unite` operation should then update this property accordingly. *RootsAreOrdered* does not require updating since it already contains the `first` encountered vertices from each *Live* partition. *ConnectedPartitions* may also receive a similar update, now using the `ExPathFromFirst` and `ExPathToFirst` instead of `ExPathFromRep` and `ExPathToRep` respectively.

Listing 7.2: Updated HistoryRepresentedByRoots definition

```
static pure boolean HistoryRepresentedByRoots(int N, seq<int> H,
    seq<int> R, seq<int> S) =
    (|H| == 0 && |R| == 0)
    ||
    (|H| > 0 && |R| > 0 &&
        HistoryRepresentedByRoots_NonEmpty(N, H, R, S));

//changed: rep(N, S, v) == top(N, R) --->
//         rep(N, S, v) == rep(N, S, top(N, R))
static pure boolean HistoryRepresentedByRoots_NonEmpty(int N,
    seq<int> H, seq<int> R, seq<int> S) =
    (\let int v = last(H); rep(N, S, v) == rep(N, S, top(N, R)) && (
        v == rep(N, S, v)
            ? ((|R| >= 2 && HistoryRepresentedByRoots_NonEmpty(N,
            init(H), getFst(pop(N, R)), S)) || (H == [v] && R == [v]))
            : (|H| > |R| && (\let seq<int> initH = init(H);
                rep(N, S, v) == rep(N, S, last(initH)) &&
                HistoryRepresentedByRoots_NonEmpty(N, initH, R, S))))
    );
```

In Listing 7.2 we show the updated definition for *HistoryRepresentedByRoots* taking this extra level of indirection into account. Most importantly it now ensures that $S^N[v] = S^N[R_{top}]$.

What is more, in the parallel algorithm the union-find structure S is implemented using a tree of representatives, as well as a cyclic list per partition. One may want to verify that each vertex in the list is indeed represented by the same root as the other vertices in the same list:

$$\forall s \in list.\, Find(list[0]) = Find(s)$$

Additionally, when a vertex in the cyclic list is *Done*, then either its partitions is already *Explored*, or it has yet to be removed from the list:

$$\forall s.\,(s.list\_status = Done) \implies (Find(s).uf\_status = Explored \vee UF[s].next.list\_status = Busy)$$

Bibliography

[1] Jiri Barnat et al. “Parallel Model Checking Algorithms for Linear-Time Temporal Logic”. In: *Handbook of Parallel Constraint Reasoning*. Ed. by Youssef Hamadi and Lakhdar Sais. Cham: Springer International Publishing, 2018, pp. 457–507. ISBN: 978-3-319-63516-3. DOI: 10.1007/978-3-319-63516-3_12. URL: https://doi.org/10.1007/978-3-319-63516-3_12.

[2] Vincent Bloemen. “Strong Connectivity and Shortest Paths for Checking Models”. English. PhD thesis. Netherlands: University of Twente, July 2019. ISBN: 978-90-365-4786-4. DOI: 10.3990/1.9789036547864.

[3] Stefan Blom et al. “The VerCors Tool Set: Verification of Parallel and Concurrent Software”. In: *IFM*. Vol. 10510. Lecture Notes in Computer Science. Springer, 2017, pp. 102–110. URL: https://link.springer.com/chapter/10.1007/978-3-319-66845-1_7.

[4] *Boogie*. https://github.com/boogie-org/boogie. Accessed: 2022-11-30.

[5] Michael Brinkmeier. “Distributed Calculation of PageRank Using Strongly Connected Components”. In: *Innovative Internet Community Systems*. Ed. by Alain Bui et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 29–40. ISBN: 978-3-540-33974-8.

[6] *Carbon: Verification-condition-generation-based verifier for the Viper intermediate verification language*. https://github.com/viperproject/carbon. Accessed: 2022-11-30.

[7] R.W. Floyd. “Assigning meaning to programs, Prec”. In: *Symposium Applied Mathematics—1967*. 1967.

[8] Harold N. Gabow. “Path-based depth-first search for strong and biconnected components”. In: *Information Processing Letters* 74.3 (2000), pp. 107–114. ISSN: 0020-0190. DOI: https://doi.org/10.1016/S0020-0190(00)00051-X. URL: https://www.sciencedirect.com/science/article/pii/S002001900000051X.

[9] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[10] J.P. Hollander. *VerCors set-comprehension bug*. 2020. URL: https://github.com/utwente-fmt/vercors/issues/557#issue-726415736 (visited on 11/02/2022).

[11] J.P. Hollander. *Verification of a model checking algorithm in VerCors*. Aug. 2021. URL: http://essay.utwente.nl/88268/.

[12] *Isabelle*. https://isabelle.in.tum.de/. Accessed: 2023-01-17.

[13] *Java Modeling Language*. https://jmlspecs.org/. Accessed: 2022-11-22.

[14] Bengt Jonsson and Tsay Yih-Kuen. “Assumption/guarantee specifications in linear-time temporal logic”. In: *Theoretical Computer Science* 167.1 (1996), pp. 47–72. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(96)00069-2. URL: https://www.sciencedirect.com/science/article/pii/0304397596000692.

[15] Peter Lammich. “Verified Efficient Implementation of Gabow’s Strongly Connected Component Algorithm”. In: *Interactive Theorem Proving*. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 325–340. ISBN: 978-3-319-08970-6.

[16] *LTSmin: Model Checking and Minimization of Labelled Transition Systems*. http://ltsmin.utwente.nl/. Accessed: 2023-02-15.

[17] Ian Munro. "Efficient determination of the transitive closure of a directed graph". In: *Information Processing Letters* 1.2 (1971), pp. 56–58. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(71)90006-8. URL: https://www.sciencedirect.com/science/article/pii/0020019071900068.

[18] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. Oct. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[19] Jaco van de Pol. "Exploring a Parallel SCC Algorithm: Using TLA+and the TLC Model Checker". In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22–30, 2022, Proceedings, Part I*. Rhodes, Greece: Springer-Verlag, 2022, pp. 535–555. ISBN: 978-3-031-19848-9. DOI: 10.1007/978-3-031-19849-6_30. URL: https://doi.org/10.1007/978-3-031-19849-6_30.

[20] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. July 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[21] Ö.F.O. Şakar. *Extending support for axiomatic data types in VerCors*. Apr. 2020. URL: http://essay.utwente.nl/80892/.

[22] *SetBasedSCC*. https://github.com/HanHollander/SetBasedSCC. Accessed: 2022-12-20.

[23] *Silicon: A Viper Verifier Based on Symbolic Execution*. https://github.com/viperproject/silicon. Accessed: 2022-11-30.

[24] *Silver: Definition of the Viper intermediate verification language*. https://github.com/viperproject/silver. Accessed: 2022-11-30.

[25] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[26] *TemplateEngine.java*. https://github.com/Jankoekenpan/vercors/blob/fa8fcc53382bccccf88d4 examples/scc/delivery/TemplateEngine.java. Accessed: 2023-01-09.

[27] *The Z3 Theorem Prover*. https://github.com/Z3Prover/z3. Accessed: 2022-11-30.

[28] Vincent Trélat and Stephan Merz. "Correctness of a Set-based Algorithm for Computing Strongly Connected Components of a Graph". In: *Archive of Formal Proofs* (Aug. 2022). https://isa-afp.org/entries/SCC_Bloemen_Sequential.html, Formal proof development. ISSN: 2150-914x.

[29] *VerCors wiki - Triggers*. https://github.com/utwente-fmt/vercors/wiki/Specification-Syntax#triggers. Accessed: 2023-01-11.

A. Uniting partitions

We use the following method to unite two partitions:

Listing A.1: UNITEROOTS adapted from Hollander [11]

```
1  requires ru(v, N);
2  requires ru(w, N);
3  requires UnionFind(N, S);
4  requires v == rep(N, S, v);
5  requires w == rep(N, S, w);
6  ensures \result == S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
7  ensures \result == S[w -> v];
8  ensures UnionFind(N, \result);
9  // lemma 7 and lemma 8, resp.:
10 ensures (\forall int i; ru(i, N) && rep(N, S, w) == {:rep(N, S, i):};
       rep(N, \result, i) == rep(N, S, v));
11 ensures (\forall int i; ru(i, N) && rep(N, S, w) != {:rep(N, S, i):};
       rep(N, \result, i) == rep(N, S, i));
12 static seq<int> uniteRoots(int N, seq<int> S, int v, int w) {
13     seq<int> T = S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
14     if (rep(N, S, w) == rep(N, S, v)) {
15         assert T == S;  // Trivial case
16     } else {
17         lemma_7_uf(N, S, T, v, w);
18         lemma_8_uf(N, S, T, v, w);
19     }
20     return T;
21 }
```

In fact, we use the exact same implementation as Hollander's original implementation [22], but added extra clauses to the contract specifying that v and w are representatives themselves already. This allows VerCors to prove the postcondition on line 7 stating that the result is actually just S with one value updated. This fact helps the verifier get through the rest of the proof. Below Hollander's original implementation is listed:

Listing A.2: UNITE operation implemented by Hollander [11]

```
requires ru(v, N);
requires ru(w, N);
requires UnionFind(N, S);
ensures \result == S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
ensures UnionFind(N, \result);
// lemma 7 and lemma 8, resp.:
ensures (\forall int i; ru(i, N) && rep(N, S, w) == {:rep(N, S, i):};
    rep(N, \result, i) == rep(N, S, v));
ensures (\forall int i; ru(i, N) && rep(N, S, w) != {:rep(N, S, i):};
    rep(N, \result, i) == rep(N, S, i));
static seq<int> unite(int N, seq<int> S, int v, int w) {
    seq<int> T = S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
```

```
    if (rep(N, S, w) == rep(N, S, v)) {
        assert T == S;  // Trivial case
    } else {
        lemma_7_uf(N, S, T, v, w);
        lemma_8_uf(N, S, T, v, w);
    }
    return T;
}
```

Listing A.3: LEMMA_7_UF by Hollander [11]

```
// Lemma 7 - Given union-find S and sequence T, where T ==
    S[SB.rep(N, S, w) -> SB.rep(N, S, v)], then
//           (\forall int i; ru(i, N) && rep(N, S, w) == rep(N, S, i);
    rep(N, T, i) == rep(N, S, v)).
//           (Proof by exhaustive contradiction)
requires ru(v, N);
requires ru(w, N);
requires UnionFind(N, S);
requires T == S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
ensures (\forall int i; ru(i, N) && rep(N, S, w) == rep(N, S, i);
    rep(N, T, i) == rep(N, S, v));
static void lemma_7_uf(int N, seq<int> S, seq<int> T, int v, int w) {
    int x = 0;

    loop_invariant rui(x, N);
    loop_invariant (\forall int i; ru(i, x) && rep(N, S, w) == rep(N, S,
         i);
        rep(N, T, i) == rep(N, S, v));
    while (x < N) {
        if (rep(N, S, w) == rep(N, S, x)) {
            if (rep(N, T, x) == rep(N, S, v)) {
                // OK
            } else {
                // rep(N, T, x) != rep(N, S, v)
                // Assertion: rep(N, T, x) != rep(N, S, v)
                // Contradiction: rep(N, T, x) == rep(N, S, v)

                int y;
                seq<int> P;

                lemma_9_uf(N, S, x) with {P = P;};
                // ExSeqPath(N, S, x, rep(N, S, x), 2) and |P| >= 2

                y = 0;
                // SeqPath(N, T, x, P[y], P[0..(y + 1)])
                loop_invariant 0 <= y && y < |P|;
                loop_invariant (\forall int z; z in P[0..y]; S[z] == T[z
```

```
                    ]);
                loop_invariant SeqPath(N, T, x, P[y], P[0..(y + 1)]);
                while (P[y] != rep(N, S, w)) {
                    y++;
                }

                // because T[rep(N, S, w)] == rep(N, S, v)
                assert SeqPath(N, T, x, rep(N, S, v), P[0..(y + 1)] +
                    [rep(N, S, v)]);  // Explicit witness
                lemma_10_uf(N, T, x, rep(N, S, v));
                // rep(N, T, x) == rep(N, S, v)

                assert false;
            }
        }
        x++;
    }
}
```

Listing A.4: LEMMA_8_UF by Hollander [11]

```
// Lemma 8 - Given union-find S and sequence T, where T ==
    S[SB.rep(N, S, w) -> SB.rep(N, S, v)], then
//           (\forall int i; ru(i, N) && rep(N, S, w) != rep(N, S, i);
    rep(N, T, i) == rep(N, S, i)).
//           (Proof by exhaustive contradiction)
requires ru(v, N);
requires ru(w, N);
requires UnionFind(N, S);
requires T == S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
ensures (\forall int i; ru(i, N) && rep(N, S, w) != rep(N, S, i);
    rep(N, T, i) == rep(N, S, i));
static void lemma_8_uf(int N, seq<int> S, seq<int> T, int v, int w) {
    int x = 0;

    loop_invariant rui(x, N);
    loop_invariant (\forall int i; ru(i, x) && rep(N, S, w) != rep(N, S,
         i);
        rep(N, T, i) == rep(N, S, i));
    while (x < N) {
        if (rep(N, S, w) != rep(N, S, x)) {
            if (rep(N, T, x) == rep(N, S, x)) {
                // OK
            } else {
                // rep(N, T, x) != rep(N, S, x)
                // Assertion: rep(N, T, x) != rep(N, S, x)
                // Contradiction: rep(N, T, x) == rep(N, S, x)

```

```
                int y;
                seq<int> P;

                lemma_9_uf(N, S, x) with {P = P;};
                // ExSeqPath(N, S, x, rep(N, S, x), 2) and |P| >= 2
                lemma_11_uf(N, S, x, P);
                // (\forall int i; i in P; rep(N, S, i) == rep(N, S, x))
                assert (\forall int i; 0 <= i && i < |P|;
                    (\forall int j; j in P[0..i]; j in P));
                // Explicit assert needed (lemma 9)
                y = 0;
                // SeqPath(N, T, x, P[y], P[0..(y + 1)])
                loop_invariant 0 <= y && y < |P|;
                loop_invariant (\forall int z; z in P[0..y]; S[z] == T[z
                    ]);
                loop_invariant SeqPath(N, T, x, P[y], P[0..(y + 1)]);
                while (P[y] != rep(N, S, x)) {
                    y++;
                }

                lemma_10_uf(N, T, x, rep(N, S, x));
                // rep(N, T, x) == rep(N, S, x)

                assert false;
            }
        }
        x++;
    }
}
```

Listing A.5: LEMMA_9_UF by Hollander [11]

```
// Lemma 9 - Given a union-find S and a state v, then there exists a
    path
// from v to the representative of v in S, rep(N, S, v). (Proof by
    induction)
yields seq<int> P;
requires ru(v, N);
requires UnionFind(N, S);
ensures SeqPath(N, S, v, rep(N, S, v), P) && |P| >= 2;
ensures ExSeqPath(N, S, v, rep(N, S, v), 2);
static void lemma_9_uf(int N, seq<int> S, int v) {
    if (S[v] == v) {
        // Base case: P = [v];
        P = [v, v];
    } else {
        // S[v] != v
        // Induction hypothesis: ExSeqPath(N, S, S[v], rep(N, S, S[v]),
```

```
            1)
        lemma_9_uf(N, S, S[v]) with {P = P;};
        P = [v] + P;

    }
}
```

Listing A.6: LEMMA_10_UF by Hollander [11]

```
// Lemma 10 - Given a sequence S and states v and r, where S[r] == r and
// there exists a path from v to r, then the representative of v in S,
// rep(N, S, v), is equal to r. (Proof by contradiction)
requires ru(v, N) && ru(r, N);
requires UnionFind(N, S);
requires S[r] == r;
requires ExSeqPath(N, S, v, r, 2);
ensures rep(N, S, v) == r;
static void lemma_10_uf(int N, seq<int> S, int v, int r) {
    if (rep(N, S, v) == r) {
        // OK
    } else {
        // rep(N, S, v) != r
        // Assertion: rep(N, S, r) == r
        //     (property of both rep() and UnionFind())
        // Contradiction: rep(N, S, r) != r
        //     (by deconstructing the path from v to r)
        int w = v;
        loop_invariant ru(w, N);
        loop_invariant rep(N, S, w) != r;
        while (w != r) { w = S[w]; }
        assert false;
    }
}
```

Listing A.7: LEMMA_11_UF by Hollander [11]

```
// Lemma 11 - Given a union-find S, a state v and a path P from v to
// rep(N, S, v), then all states in P also have the representative
// of v as representative in S. (Proof by induction)
requires ru(v, N);
requires UnionFind(N, S);
requires SeqPath(N, S, v, rep(N, S, v), P) && |P| >= 2;
ensures (\forall int i; i in P; rep(N, S, i) == rep(N, S, v));
static void lemma_11_uf(int N, seq<int> S, int v, seq<int> P) {
    if (|P| == 2) {
        // Base case: v == rep(N, S, v)
    } else {
        // |P| > 2
```

```
        // Induction hypothesis: (\forall int i; i in tail(P);
        //     rep(N, S, i) == rep(N, S, S[v]))
        lemma_11_uf(N, S, S[v], tail(P));
        assert P == [v] + tail(P);  // Explicit assert needed
    }
}
```

B. Proving transitivity of $ExFittingPath$

We provide the following PVL proof to prove that in fact, concatenating two fitting paths ($x \xrightarrow[C1]{}{}^{*} y$ and $y \xrightarrow[C2]{}{}^{*} z$ results in a fitting path $x \xrightarrow[CTotal]{}{}^{*} z$. This PVL prove uses $ExFittingPath$, proving the existence of the concatenated path. In these lemmas G denotes the graph, and N denotes the number of vertices of G. C, $C1$, $C2$ and $CTotal$ are sets of vertices in which the paths are contained.

Listing B.1: ExFittingPath transitivity lemmas

```
inline static boolean Lemma_ExFittingPath_Transitivity_Pure(int N,
    seq<seq<boolean>> G, int x, int y, int z, set<int> C) =
    Lemma_ExFittingPath_Transitivity_Pure2(N, G, x, y, z, C, C, C);

requires AdjacencyMatrix(N, G);
requires ExFittingPath(N, G, x, y, 1, C1);
requires ExFittingPath(N, G, y, z, 1, C2);
requires C1 <= CTotal && C2 <= CTotal;
ensures (\exists seq<int> P; 1 <= |P|;
    FittingPath(N, G, x, y, P, C1) && (\exists seq<int> Q; 1 <= |Q|;
        FittingPath(N, G, y, z, Q, C2) &&
            FittingPath(N, G, x, z, P + tail(Q), CTotal)
    )
);
ensures \result == ExFittingPath(N, G, x, z, 1, CTotal);
static boolean Lemma_ExFittingPath_Transitivity_Pure2(int N,
    seq<seq<boolean>> G, int x, int y, int z, set<int> C1, set<int> C2,
    set<int> CTotal) = true;
```

C. Proving *HistoryRepresentedByRoots* for StackCollapse

Here we present a lemma for proving that *HistoryRepresentedByRoots* (Definition 8) still holds after one call to `uniteRoots` in StackCollapse (Listing 3.18, line 7). The lemma itself is recursive and calls into two other lemmas, but we guarantee termination because $|H|$ is decreased with every lemma call.

Listing C.1: Lemma for proving $HistoryRepresentedByRoots\_NonEmpty(N, H, R, S)$ still holds after one `uniteRoots`.

```
requires History(N, H) && Stack(N, oldR) && Stack(N, newR) && UnionFind(N, oldS) && UnionFind(N, newS);
requires |oldR| >= 2 && |H| >= |oldR|;
requires newR == getFst(pop(N, oldR));
requires (\forall int r; r in oldR; rep(N, oldS, r) == r);
requires newS == oldS[SB.top(N, oldR) -> SB.top(N, newR)];
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, top(N, newR)));
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires HistoryRepresentedByRoots4_NonEmpty(N, H, oldR, oldS);
ensures HistoryRepresentedByRoots4_NonEmpty(N, H, newR, newS);
static void Lemma_HistoryRepresentedByRoots4_NonEmpty_StackCollapse(
    int N, seq<int> H, seq<int> oldR, seq<int> oldS, seq<int> newR, seq<int> newS) {

    //all roots are still the same as before, and they are still their own reps.
    assert (\forall int i; 0 <= i && i < |newR|-1;
        rep(N, oldS, newR[i]) == rep(N, newS, newR[i]) &&
        rep(N, newS, newR[i]) == newR[i]
    );

    int v = last(H);
    assert rep(N, newS, v) == top(N, newR);
    assert v != rep(N, newS, v);
    assert v != top(N, newR);

    //  What should the proof look like?
    //  we should recurse over the invariant as long as |R| >= 2
    //  and prove HistoryRepresentedByRoots4_NonEmpty(N, init(H), newR, newS)
    //  and then combine that with rep(N, newS, v) == top(N, newR) in order to prove
    //  HistoryRepresentedByRoots4_NonEmpty(N, H, newR, newS).

```

```
    if (|H| > |oldR|) {

        if (HistoryRepresentedByRoots4_NonEmpty(N, init(H), oldR, oldS)) {
            Lemma_HistoryRepresentedByRoots4_NonEmpty_StackCollapse(N, init(H), oldR, oldS, newR, newS);
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), newR, newS);
            assert |H| > |newR|;
            assert HistoryRepresentedByRoots4_NonEmpty(N, H, newR, newS);
        } else {
            assert !HistoryRepresentedByRoots4_NonEmpty(N, init(H), oldR, oldS);

            if (rep(N, oldS, v) == v) {
                assert v == top(N, oldR);
                assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), getFst(pop(N, oldR)), oldS);
                assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), newR, oldS);
                Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_1(N, init(H), top(N, oldR), top(N, newR), oldS, newS, newR);
                assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), newR, newS);
                assert HistoryRepresentedByRoots4_NonEmpty(N, H, newR, newS);
            } else {
                assert rep(N, oldS, v) != v;
                assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), oldR, oldS); //contradicts !HistoryRepresentedByRoots4_NonEmpty(N, init(H), oldR, oldS);
                assert false;
            }
        }

    } else {
        assert |H| == |oldR|;
        assert |H| >= 2;
        assert rep(N, oldS, v) == top(N, oldR);

        if (v == rep(N, oldS, v)) {
            assert |oldR| >= 2;
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), getFst(pop(N, oldR)), oldS);
```

```
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), newR, oldS);
            Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_3(N, init(H), top(N, oldR), top(N, newR), oldS, newS, newR);
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(H), newR, newS);
            assert |H| > |newR|;
            assert rep(N, newS, v) == top(N, newR);
            assert HistoryRepresentedByRoots4_NonEmpty(N, H, newR, newS);
        } else {
            assert |H| > |oldR|;    //contradicts |H| == |oldR|;
            assert false;
        }
    }

    assert HistoryRepresentedByRoots4_NonEmpty(N, H, newR, newS);
}

requires History(N, iH) && ru(oldTop, N) && ru(newTop, N) && UnionFind(N, oldS) && UnionFind(N, newS) && Stack(N, theR);
requires |iH| > |theR| && |iH| > 0 && |theR| > 0;
requires oldTop == rep(N, oldS, oldTop);
requires newTop == rep(N, newS, newTop);
requires newS == oldS[oldTop -> newTop];
requires (\forall int i; ru(i, N) && rep(N, oldS, oldTop) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, newTop));
requires (\forall int i; ru(i, N) && rep(N, oldS, oldTop) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires (\forall int r; r in theR; rep(N, oldS, r) == r);
requires (\forall int r; r in theR; rep(N, newS, r) == r);
requires HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, oldS);
ensures HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
static void Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_1
    (int N, seq<int> iH, int oldTop, int newTop, seq<int> oldS, seq<int> newS, seq<int> theR) {

    int u = last(iH);

    //why is historyRepresentedByRootsLast3(N, initH, theR, newS) true?
    //  because S still points to the same reps for all possible roots!!
    //  so we just unfold, and perform a recursive call :)

    assert |iH| >= 2;

    if (u == rep(N, oldS, u)) {
        assert u == top(N, theR);
        assert |theR| >= 2;
```

```
        assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), getFst(pop(N, theR)), oldS);
        Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_1(N, init(iH), oldTop, newTop, oldS, newS, getFst(pop(N, theR)));
        assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), getFst(pop(N, theR)), newS);
        assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
    } else {
        assert u != rep(N, oldS, u);
        assert |iH| > |theR|;
        assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), theR, oldS);
        Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_2(N, init(iH), oldTop, newTop, oldS, newS, theR);
        assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), theR, newS);
        assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
    }

    assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
}

requires History(N, iH) && ru(oldTop, N) && ru(newTop, N) && UnionFind(N, oldS) && UnionFind(N, newS) && Stack(N, theR);
requires |iH| >= |theR| && |iH| > 0 && |theR| > 0;
requires oldTop == rep(N, oldS, oldTop) && newTop == rep(N, newS, newTop);
requires newS == oldS[oldTop -> newTop];
requires (\forall int i; ru(i, N) && rep(N, oldS, oldTop) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, newTop));
requires (\forall int i; ru(i, N) && rep(N, oldS, oldTop) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires (\forall int r; r in theR; rep(N, oldS, r) == r);
requires (\forall int r; r in theR; rep(N, newS, r) == r);
requires HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, oldS);
ensures HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
static void Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_2
    (int N, seq<int> iH, int oldTop, int newTop, seq<int> oldS, seq<int> newS, seq<int> theR) {

    //why is historyRepresentedByRootsLast3(N, iH, theR, newS) true?
    //  because S still points to the same reps for all possible roots!!
    //  so we just unfold, and perform a recursive call :)

    int u = last(iH);

    //unfold:
    if (u == top(N, theR)) {
```

```
        if (|iH| == 1 && |theR| == 1) {
            assert [u] == iH;
            assert [u] == theR;
            assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
        } else {
            assert |iH| >= 2;
            assert |theR| >= 2;
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), getFst(pop(N, theR)), oldS);
            if (|iH| > |theR|) {
                Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_1(N, init(iH), oldTop, newTop, oldS, newS, getFst(pop(N, theR)));
                assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), getFst(pop(N, theR)), newS);
                assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
            } else {
                assert |iH| == |theR|;
                Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_2(N, iH, oldTop, newTop, oldS, newS, theR);
                assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
            }
        }
    } else {
        assert u != top(N, theR);
        assert |iH| >= 2;
        assert |iH| > |theR|;
        assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), theR, oldS);
        assert u != rep(N, oldS, u);
        Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_2(N, init(iH), oldTop, newTop, oldS, newS, theR);
        assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), theR, newS);
        assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
    }

    assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
}

requires History(N, iH) && ru(oldTop, N) && ru(newTop, N) && UnionFind(N, oldS) && UnionFind(N, newS) && Stack(N, theR);
requires |iH| == |theR| && |iH| > 0 && |theR| > 0;
requires oldTop == rep(N, oldS, oldTop) && newTop == rep(N, newS, newTop);
```

```
requires newS == oldS[oldTop -> newTop];
requires (\forall int i; ru(i, N) && rep(N, oldS, oldTop) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, newTop));
requires (\forall int i; ru(i, N) && rep(N, oldS, oldTop) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires (\forall int r; r in theR; rep(N, oldS, r) == r);
requires (\forall int r; r in theR; rep(N, newS, r) == r);
requires HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, oldS);
ensures HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
static void Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_3
    (int N, seq<int> iH, int oldTop, int newTop, seq<int> oldS, seq<int> newS, seq<int> theR) {

    int v = last(iH);
    assert rep(N, oldS, v) == top(N, theR);
    if (|iH| >= 2) {
        if (v == rep(N, oldS, v)) {
            assert |theR| >= 2;
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), getFst(pop(N, theR)), oldS);
            Lemma_HistoryRepresentedByRoots4_NonEmpty_unite_3(N, init(iH), oldTop, newTop, oldS, newS, getFst(pop(N, theR)));
            assert HistoryRepresentedByRoots4_NonEmpty(N, init(iH), getFst(pop(N, theR)), newS);
            assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
        } else {
            assert |iH| > |theR|; //violation of precondition |iH| == |theR|
            assert false;
        }
    }

    assert HistoryRepresentedByRoots4_NonEmpty(N, iH, theR, newS);
}
```

D. Proving *ConnectedPartitions* for StackCollapse

Here we present a dozen lemmas that were required to prove that StackCollapse preserves *ConnectedPartitions*. Apart from `Lemma_Concatenation` (which performs the final concatenation of paths), all these lemmas are called in the body of the while loop, in order to proof preservation of some loop invariants. These loop invariants correspond to the postconditions of the lemmas.

Listing D.1: Lemmas for proving $ConnectedPartitions(N, G, R, S)$ at StackCollapse

```
requires AdjacencyMatrix(N, G) && UnionFind(N, oldS) && UnionFind(N,
    newS) && Stack(N, oldR) && Stack(N, newR) && ru(v, N) && ru(w, N);
requires G[v][w];
requires |oldR| > 0 && |newR| > 0;
requires ConnectedPartitions2(N, G, oldS);
requires RootPath2(N, G, oldR, oldS);
requires top(N, oldR) == rep(N, oldS, v) && top(N, newR) == rep(N, newS,
     v);
requires rep(N, oldS, w) == rep(N, newS, w) && rep(N, newS, w) == top(N,
     newR);
requires (\forall int x; ru(x, N) && rep(N, newS, x) == top(N, newR);
    ExFittingPath(N, G, rep(N, oldS, x), top(N, oldR), 1, part(N, newS,
    top(N, newR))));
requires (\forall int x; ru(x, N) && rep(N, newS, x) == top(N, newR);
    part(N, oldS, x) <= part(N, newS, top(N, newR)));
ensures (\forall int x; ru(x, N) && rep(N, newS, x) == top(N, newR);
    ExPathToRep(N, G, newS, x));
static void Lemma_Concatenation(int N, seq<seq<boolean>> G,
        seq<int> oldS, seq<int> newS,
        seq<int> oldR, seq<int> newR,
        int v, int w) {

        loop_invariant rui(i, N);
        loop_invariant (\forall int j; 0 <= j && j < i && rep(N, newS, j
            ) == top(N, newR); ExPathToRep(N, G, newS, j));
        for (int i = 0; i < N; i++) {
                if (rep(N, newS, i) == top(N, newR)) {
                        //concat:
                        //i ~> \old(rep(i)) ~> \old(rep(v)) ~> v -> w ~>
                            \old(rep(w)) = rep(w) = rep(v) = rep(i)

                        // i ~> old(rep(i))
                        assert CP(N, G, oldS, i);
                        assert ExPathToRep(N, G, oldS, i);
                        set<int> C1 = part(N, oldS, i);
                        assert ExFittingPath(N, G, i, rep(N, oldS, i),
                            1, C1);
```

```
// old(rep(i)) ~> old(rep(v))
assert ExFittingPath(N, G, rep(N, oldS, i), top(
   N, oldR), 1, part(N, newS, top(N, newR)));
   //from precondition
set<int> C2 = part(N, newS, top(N, newR));
assert ExFittingPath(N, G, rep(N, oldS, i), top(
   N, oldR), 1, C2);      //substitute set
assert ExFittingPath(N, G, rep(N, oldS, i), rep(
   N, oldS, v), 1, C2);    //substitute last
   element of the path

// old(rep(v)) ~> v
assert CP(N, G, oldS, v);
assert ExPathFromRep(N, G, oldS, v);
set<int> C3 = part(N, oldS, v);
assert ExFittingPath(N, G, rep(N, oldS, v), v,
   1, C3);

// v -> w
set<int> C4 = {v, w};
assert FittingPath(N, G, v, w, [v, w], C4);
assert ExFittingPath(N, G, v, w, 1, C4);

// w ~> old(rep(w))
assert CP(N, G, oldS, w);
assert ExPathToRep(N, G, oldS, w);
set<int> C5 = part(N, oldS, w);
assert ExFittingPath(N, G, w, rep(N, oldS, w),
   1, C5);

//prove subsets!
assert C1 <= C2;
assert C3 <= C2;
assert C4 <= C2;
assert C5 <= C2;

//concat all the paths!
assert Lemma_ExFittingPath_Transitivity_Pure2(N,
    G, i, rep(N, oldS, i), rep(N, oldS, v), C1,
    C2, C2);
assert Lemma_ExFittingPath_Transitivity_Pure2(N,
    G, i, rep(N, oldS, v), v, C2, C3, C2);
assert Lemma_ExFittingPath_Transitivity_Pure2(N,
    G, i, v, w, C2, C4, C2);
assert Lemma_ExFittingPath_Transitivity_Pure2(N,
    G, i, w, rep(N, oldS, w), C2, C5, C2);
```

```
            //final substitutions!
            assert part(N, newS, top(N, newR)) == part(N, newS, i);
            assert ExFittingPath(N, G, i, rep(N, newS, w), 1, part(N, newS, i));
            assert ExFittingPath(N, G, i, rep(N, newS, i), 1, part(N, newS, i));

            //conclusion
            assert ExPathToRep(N, G, newS, i);
        }
    }
}

requires UnionFind(N, oldS) && UnionFind(N, newS) && UnionFind(N, originalS)
    && Stack(N, oldR) && Stack(N, newR) && Stack(N, originalR)
    && ru(rRecent, N) && ru(rOld, N);
requires rep(N, oldS, rRecent) == rRecent && rep(N, oldS, rOld) == rOld && rRecent != rOld; //last proposition is redundant
requires newS == oldS[rRecent -> rOld];
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, rOld));
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires |oldR| >= 2 && Prefix(oldR, originalR) && newR == getFst(pop(N, oldR)) && Prefix(newR, originalR); //last proposition is redundant
requires rRecent == top(N, oldR) && rOld == top(N, newR);
requires (\forall int i; |oldR| - 1 <= i && i < |originalR|; {:part(N, originalS, originalR[i]):} <= part(N, oldS, rRecent));   // don't _need_ this explicit trigger
requires part(N, originalS, rOld) == part(N, oldS, rOld);
ensures (\forall int i; |newR| - 1 <= i && i < |originalR|; part(N, originalS, originalR[i]) <= part(N, newS, rOld));
static void Lemma_AllUnitedPartsAreSubsetOfPartV_Maintained(int N,
    seq<int> oldS, seq<int> newS, seq<int> originalS,
    seq<int> oldR, seq<int> newR, seq<int> originalR,
    int rRecent, int rOld) {

    set<int> mergedPartition = part(N, newS, rOld);
    assert mergedPartition == part(N, oldS, rRecent) + part(N, oldS, rOld);

    assert part(N, oldS, rRecent) <= part(N, newS, rOld);

    loop_invariant |oldR| - 1 <= i && i <= |originalR|;
    loop_invariant (\forall int j; |oldR| - 1 <= j && j < i; part(N, originalS, originalR[j]) <= part(N, newS, rOld));
```

```
        for (int i = |oldR| - 1; i < |originalR|; i++) {
                assert part(N, originalS, originalR[i]) <= part(N, oldS, rRecent);   //precondition
                assert part(N, oldS, rRecent) <= part(N, newS, rOld);                    //result of 'unite' (already proven above)
                assert part(N, originalS, originalR[i]) <= part(N, newS, rOld);      //transitivity of subset relation
        }

        assert (\forall int i; |oldR| - 1 <= i && i < |originalR|; part(N, originalS, originalR[i]) <= part(N, newS, rOld));
        assert |newR| == |oldR| - 1;
        assert part(N, originalS, originalR[|newR|-1]) <= part(N, newS, rOld);
        assert (\forall int i; |newR| - 1 <= i && i < |originalR|; part(N, originalS, originalR[i]) <= part(N, newS, rOld));
}

requires UnionFind(N, oldS) && UnionFind(N, newS) && UnionFind(N, originalS)
        && Stack(N, oldR) && Stack(N, newR) && Stack(N, originalR)
        && ru(rRecent, N) && ru(rOld, N);
requires rep(N, oldS, rRecent) == rRecent && rep(N, oldS, rOld) == rOld && rRecent != rOld; //last proposition is redundant
requires newS == oldS[rRecent -> rOld];
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, rOld));
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires |oldR| >= 2 && Prefix(oldR, originalR) && newR == getFst(pop(N, oldR)) && Prefix(newR, originalR); //last proposition is redundant
requires rRecent == top(N, oldR) && rOld == top(N, newR);
requires (\forall int i; 0 <= i && i < |oldR|; oldR[i] == rep(N, oldS, oldR[i]));
requires (\forall int i; 0 <= i && i < |oldR| - 1; {:part(N, originalS, originalR[i]):} == part(N, oldS, oldR[i]));            // don't _need_ this explicit trigger.
ensures (\forall int i; 0 <= i && i < |newR| - 1; part(N, originalS, originalR[i]) == part(N, newS, newR[i]));
static void Lemma_AllUnunitedPartsRemainTheSame(int N,
        seq<int> oldS, seq<int> newS, seq<int> originalS,
        seq<int> oldR, seq<int> newR, seq<int> originalR,
        int rRecent, int rOld) {

        assert (\forall int i; 0 <= i && i < |newR|;
                rep(N, oldS, newR[i]) == rep(N, newS, newR[i]) &&
                rep(N, newS, newR[i]) == newR[i]
```

```
	);

	int i = 0;

	loop_invariant 0 <= i && i <= |newR| - 1;
	loop_invariant (\forall int j; 0 <= j && j < i; part(N, originalS, originalR[j]) == part(N, newS, newR[j]));
	while (i < |newR| - 1) {
		int r = newR[i];
		assert r == originalR[i] && r == oldR[i];
		assert r != rRecent && r != rOld;

		assert rep(N, newS, r) == rep(N, oldS, r);                    //precondition (all elements in R are their own rep)
		assert part(N, originalS, r) == part(N, oldS, r);             //precondition (instantiate \forall, substitute originalR[i] with r, and oldR[i] with r)
		assert part(N, oldS, r) == part(N, newS, r);                  //by 'unite'
		assert part(N, originalS, r) == part(N, newS, r);             //by transitivity of '=='
		assert part(N, originalS, originalR[i]) == part(N, newS, newR[i]);      //substitute r for originalR[i] and newR[i].
		assert (\forall int j; 0 <= j && j <= i; part(N, originalS, originalR[j]) == part(N, newS, newR[j])); //unify //does work in this file!
		i++;
		assert (\forall int j; 0 <= j && j < i; part(N, originalS, originalR[j]) == part(N, newS, newR[j]));
	}

	assert (\forall int i; 0 <= i && i < |newR| - 1; part(N, originalS, originalR[i]) == part(N, newS, newR[i]));
}

requires AdjacencyMatrix(N, G);
requires UnionFind(N, oldS);
requires UnionFind(N, newS);
requires ru(rRecent, N);
requires ru(rOld, N);
requires newS == oldS[rRecent -> rOld];
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, rOld));
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
```

```
requires rep(N, oldS, rRecent) == rRecent;
requires rep(N, oldS, rOld) == rOld;
requires (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; {:CP(N, G, oldS, x):});
ensures (\forall int x; ru(x, N) && rep(N, newS, x) != rOld; {:CP(N, G, newS, x):});
static void Lemma_CP_x_not_in_part_V(int N, seq<seq<boolean>> G, seq<int> oldS, seq<int> newS, int rRecent, int rOld) {

        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; part(N, oldS, x) <= part(N, newS, x));

        //reps are still the same!
        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; rep(N, oldS, x) == rep(N, newS, x));
        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; !(x in part(N, oldS, rRecent)));

        //if x is not in v's partition now, it also wasn't before.
        assert (\forall int x; ru(x, N) && !(x in part(N, newS, rOld)); !(x in part(N, oldS, rRecent)));
        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; rep(N, oldS, x) == rep(N, newS, x));

        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent;
                Lemma_CP_Maintained_Pure(N, G, oldS, newS, x)
        );

        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; {:CP(N, G, newS, x):});

        assert (\forall int x; ru(x, N) && rep(N, newS, x) != rOld; {:CP(N, G, newS, x):});
}

requires AdjacencyMatrix(N, G) && UnionFind(N, oldS) && UnionFind(N, newS) && ru(rRecent, N) && ru(rOld, N);
requires rep(N, oldS, rRecent) == rRecent && rep(N, oldS, rOld) == rOld && rRecent != rOld;
requires newS == oldS[rRecent -> rOld];
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, rOld));
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires ExFittingPath(N, G, rOld, rRecent, 1, part(N, oldS, rOld) + {rRecent});              //by RootPath // change to "RootPath2(N, G, oldR, oldS)" maybe?
```

```
requires (\forall int x; ru(x, N) && rep(N, oldS, x) == rRecent; ExPathFromRep(N, G, oldS, x)); //by CP
requires (\forall int x; ru(x, N) && rep(N, oldS, x) == rOld; CP(N, G, oldS, x));              //by CP // change to "rep(N, oldS, x) != rRecent" maybe?
ensures (\forall int x; ru(x, N) && rep(N, newS, x) == rOld; ExPathFromRep(N, G, newS, x));
static void Lemma_RepXToX_Maintained(int N, seq<seq<boolean>> G, seq<int> oldS, seq<int> newS, int rRecent, int rOld) {
        assert (\forall int i; ru(i, N) && {:rep(N, oldS, i):} == rOld; rep(N, newS, i) == rOld);

        set<int> newPartV = part(N, oldS, rRecent) + part(N, oldS, rOld);
        assert part(N, newS, rOld) == newPartV;

        assert ExFittingPath(N, G, rOld, rRecent, 1, part(N, oldS, rOld) + {rRecent});

        int i = 0;
        loop_invariant rui(i, N);
        loop_invariant (\forall int x; ru(x, i) && rep(N, newS, x) == rOld; ExPathFromRep(N, G, newS, x));
        while (i < N) {
                if (rep(N, newS, i) == rOld) {
                        assert rep(N, oldS, i) == rRecent || rep(N, oldS, i) == rOld;
                        if (rep(N, oldS, i) == rOld) {
                                //old rep of i is rOld. Re-use existing path "rOld ~> i".
                                assert rep(N, newS, i) == rep(N, oldS, i);
                                assert rep(N, newS, i) == rOld;
                                assert CP(N, G, oldS, i);
                                    //from precondition
                                assert ExPathFromRep(N, G, oldS, i);
                                    //part of definition of CP
                                assert Lemma_PathFromRep_Maintained_Pure(N, G, oldS, newS, i);
                                assert ExPathFromRep(N, G, newS, i);
                        } else {
                                //old rep of i is rRecent. So we concat the paths "rOld ~> rRecent" and "rRecent ~> i".
                                assert rep(N, oldS, i) == rRecent;
                                assert
                                    Lemma_ExFittingPath_Transitivity_Pure2(N, G, rOld, rRecent, i, part(N,
```

```
oldS, rOld) + {rRecent}, part(N, oldS, rRecent), newPartV);
                        assert ExFittingPath(N, G, rOld, i, 1, newPartV);
                        assert ExPathFromRep(N, G, newS, i);
                    }
                    assert ExPathFromRep(N, G, newS, i);
                }
                i++;
        }
}

requires AdjacencyMatrix(N, G) && Stack(N, oldR) && UnionFind(N, oldS) && Stack(N, newR) && UnionFind(N, newS);
requires |oldR| >= 2;
requires newR == getFst(pop(N, oldR));
requires ru(rRecent, N) && ru(rOld, N);
requires newS == oldS[rRecent -> rOld];
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, rOld));
requires (\forall int i; ru(i, N) && rep(N, oldS, rRecent) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
requires rep(N, oldS, rRecent) == rRecent && rep(N, oldS, rOld) == rOld;
requires rRecent == top(N, oldR) && rOld == top(N, newR);
requires RootPath2(N, G, oldR, oldS);
ensures RootPath2(N, G, newR, newS);
static void Lemma_RootPath_Maintained(int N, seq<seq<boolean>> G, seq<int> oldR, seq<int> oldS, seq<int> newR, seq<int> newS, int rRecent, int rOld) {

        assert (\forall int x; ru(x, N); part(N, oldS, x) <= part(N, newS, x));
        assert (\forall int x; ru(x, N) && rep(N, oldS, x) != rRecent; rep(N, oldS, x) == rep(N, newS, x));

        assert RootPath2(N, G, oldR[0..|oldR|-1], oldS);
        assert RootPath2(N, G, newR, oldS);
        assert (\forall int i; ru(i, |newR|-1); Lemma_RootPathPart_Maintained_Pure(N, G, oldS, newS, newR[i], newR[i+1]));
        assert RootPath2(N, G, newR, newS);
}

// Proves that for every node x where x is not represented by the new top of the stack,
//      the partition of x is in the original union-find is equal to x's current partition.
//
```

```
requires AdjacencyMatrix(N, G) && Stack(N, newR) && Stack(N, oldR);
requires UnionFind(N, newS) && UnionFind(N, oldS) && UnionFind(N, originalS);
requires |newR| > 0 && |oldR| > 0;
//
requires newR == getFst(pop(N, oldR));
requires newS == oldS[SB.top(N, oldR) -> SB.top(N, newR)];
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, top(N, newR)));
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
//
requires (\forall int i; ru(i, |oldR|); rep(N, oldS, oldR[i]) == oldR[i]);
requires (\forall int i; ru(i, |newR|); rep(N, newS, newR[i]) == newR[i]);
requires rep(N, oldS, top(N, oldR)) != rep(N, oldS, top(N, newR));
//
requires (\forall int x; ru(x, N) && rep(N, oldS, x) != top(N, oldR); part(N, originalS, x) == part(N, oldS, x));
//
ensures (\forall int x; ru(x, N) && rep(N, newS, x) != top(N, newR); part(N, originalS, x) == part(N, newS, x));
//
static void Lemma_repIsNotTop_implies_oldPartIsSame(int N, seq<seq<boolean>> G,
        seq<int> newS, seq<int> oldS, seq<int> originalS,
        seq<int> newR, seq<int> oldR) {

        int rOld = top(N, newR);
        int rRecent = top(N, oldR);

        loop_invariant rui(i, N);
        loop_invariant (\forall int j; ru(j, i) && rep(N, newS, j) != rOld; part(N, originalS, j) == part(N, newS, j));
        for (int i = 0; i < N; i++) {
                int theRep = rep(N, newS, i);
                if (theRep != rOld) {
                        assert theRep != rOld && theRep != rRecent;
                        assert rep(N, oldS, i) != rRecent;

                        assert rep(N, oldS, i) == rep(N, newS, i);
                        assert part(N, oldS, i) == part(N, newS, i);

                        assert part(N, originalS, i) == part(N, oldS, i);
                        assert part(N, originalS, i) == part(N, newS, i);
```

```
                    }
          }

          assert (\forall int x; ru(x, N) && rep(N, newS, x) != top(N, newR); part(N, originalS, x) == part(N, newS, x));
}

// Proves that for every node x where x is represented by the new top of the stack,
//       the partition of x is in the original union-find is a subset of x’s current partition.
//
requires AdjacencyMatrix(N, G) && Stack(N, newR) && Stack(N, oldR);
requires UnionFind(N, newS) && UnionFind(N, oldS) && UnionFind(N, originalS);
requires |newR| > 0 && |oldR| > 0;
//
requires newR == getFst(pop(N, oldR));
requires newS == oldS[SB.top(N, oldR) -> SB.top(N, newR)];
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, top(N, newR)));
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
//
requires (\forall int i; ru(i, |oldR|); rep(N, oldS, oldR[i]) == oldR[i]);
requires (\forall int i; ru(i, |newR|); rep(N, newS, newR[i]) == newR[i]);
requires rep(N, oldS, top(N, oldR)) != rep(N, oldS, top(N, newR));
//
requires (\forall int x; ru(x, N) && rep(N, oldS, x) != top(N, oldR); part(N, originalS, x) == part(N, oldS, x));
requires (\forall int x; ru(x, N) && rep(N, oldS, x) == top(N, oldR); part(N, originalS, x) <= part(N, oldS, top(N, oldR)));
//
ensures (\forall int x; ru(x, N) && rep(N, newS, x) == top(N, newR); part(N, originalS, x) <= part(N, newS, top(N, newR)));
//
static void Lemma_repIsTop_implies_oldPartIsSubset(int N, seq<seq<boolean>> G,
          seq<int> newS, seq<int> oldS, seq<int> originalS,
          seq<int> newR, seq<int> oldR) {

          int rOld = top(N, newR);
          int rRecent = top(N, oldR);

          set<int> newPartV = part(N, newS, rOld);
          assert newPartV == part(N, oldS, rRecent) + part(N, oldS, rOld);
```

```

        loop_invariant rui(i, N);
        loop_invariant (\forall int j; ru(j, i) && rep(N, newS, j) == top(N, newR); part(N, originalS, j) <= part(N, newS, rOld));
        for (int i = 0; i < N; i++) {
                if (rep(N, newS, i) == rOld) {
                        assert rep(N, oldS, i) == rRecent || rep(N, oldS, i) == rOld;

                        if (rep(N, oldS, i) == rOld) {
                                //old rep of i is rOld.

                                assert part(N, oldS, i) == part(N, oldS, rOld);
                                assert part(N, originalS, i) == part(N, oldS, i);
                                assert part(N, originalS, i) == part(N, oldS, rOld);

                                assert part(N, originalS, i) <= newPartV;
                        } else {
                                //old rep of i is rRecent.
                                assert rep(N, oldS, i) == rRecent;

                                assert part(N, oldS, i) == part(N, oldS, rRecent);

                                assert part(N, originalS, i) <= part(N, oldS, i);
                                assert part(N, oldS, i) <= newPartV;
                                assert part(N, originalS, i) <= newPartV;
                        }

                        assert part(N, originalS, i) <= newPartV;
                }
        }

        assert (\forall int x; ru(x, N) && rep(N, newS, x) == top(N, newR); part(N, originalS, x) <= part(N, newS, top(N, newR)));
}

// Proves that for every node x that has as its new rep the top of the stack,
//      there exists a path from rep(N, \old(S), x) to top(N, \old(R)) contained within part(N, S, top(N, R)).
```

```
//
requires AdjacencyMatrix(N, G);
requires Stack(N, originalR) && Stack(N, oldR) && Stack(N, newR);
requires UnionFind(N, originalS) && UnionFind(N, oldS) && UnionFind(N, newS);
requires |originalR| > 0 && |oldR| > 0 && |newR| > 0;
//
requires newR == getFst(pop(N, oldR));
requires Prefix(oldR, originalR);
requires newS == oldS[SB.top(N, oldR) -> SB.top(N, newR)];
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) == {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, top(N, newR)));
requires (\forall int i; ru(i, N) && rep(N, oldS, top(N, oldR)) != {:rep(N, oldS, i):}; rep(N, newS, i) == rep(N, oldS, i));
//
requires (\forall int i; ru(i, |originalR|); rep(N, originalS, originalR[i]) == originalR[i]);
requires (\forall int i; ru(i, |oldR|); rep(N, oldS, oldR[i]) == oldR[i]);
requires (\forall int i; ru(i, |newR|); rep(N, newS, newR[i]) == newR[i]);
requires rep(N, oldS, top(N, oldR)) != rep(N, oldS, top(N, newR));
requires ExFittingPath(N, G, top(N, newR), top(N, oldR), 1, part(N, oldS, top(N, newR)) + {top(N, oldR)});
//
requires (\forall int i; |newR| - 1 <= i && i < |originalR|; part(N, originalS, originalR[i]) <= part(N, newS, top(N, newR)));
requires (\forall int i; |newR| - 1 <= i && i < |originalR|; rep(N, newS, originalR[i]) == top(N, newR));
requires RootPath2(N, G, originalR, originalS);
requires (\forall int x; ru(x, N) && rep(N, oldS, x) != top(N, oldR); rep(N, oldS, x) == rep(N, originalS, x));
requires (\forall int x; ru(x, N) && rep(N, oldS, x) == top(N, oldR);
        ExFittingPath(N, G, rep(N, originalS, x), top(N, originalR), 1, part(N, oldS, top(N, oldR))));
//
ensures (\forall int x; ru(x, N) && rep(N, newS, x) == top(N, newR);
        ExFittingPath(N, G, rep(N, originalS, x), top(N, originalR), 1, part(N, newS, top(N, newR))));
//
static void Lemma_PathToOldRepV(int N, seq<seq<boolean>> G,
        seq<int> originalR, seq<int> oldR, seq<int> newR,
        seq<int> originalS, seq<int> oldS, seq<int> newS) {

        int rRecent = top(N, oldR);
        int rOld = top(N, newR);

```

```
        set<int> newPartV = part(N, oldS, rRecent) + part(N, oldS, rOld);
        assert part(N, newS, rOld) == newPartV;

        int i = 0;
        loop_invariant rui(i, N);
        loop_invariant (\forall int x; ru(x, i) && rep(N, newS, x) == rOld;
                ExFittingPath(N, G, rep(N, originalS, x), top(N, originalR), 1, part(N, newS, rOld)));
        while (i < N) {
                if (rep(N, newS, i) == rOld) {
                        assert rep(N, oldS, i) == rRecent || rep(N, oldS, i) == rOld;
                        if (rep(N, oldS, i) == rOld) {
                                //old rep of i is rOld. So we concat the paths "rOld ~> rRecent" and "rRecent ~> originalTop".
                                assert rep(N, originalS, i) == rOld;
                                assert RootPath2(N, G, originalR, originalS);

                                Lemma_RootPathToTop_All(N, G, originalR, originalS, newPartV, |newR|-1);
                                assert (\forall int j; |newR|-1 <= j && j < |originalR|; ExFittingPath(N, G, originalR[j], top(N, originalR), 1, newPartV));
                                assert ExFittingPath(N, G, rRecent, top(N, originalR), 1, newPartV);
                                assert Lemma_ExFittingPath_Transitivity_Pure2(N, G, rOld, rRecent, top(N, originalR), part(N, oldS, top(N, newR)) + {top(N, oldR)}, newPartV, newPartV);
                                assert ExFittingPath(N, G, rOld, top(N, originalR), 1, newPartV);

                                assert ExFittingPath(N, G, rep(N, originalS, i), top(N, originalR), 1, newPartV);
                        } else {
                                //old rep of i is rRecent. So we re-use the existing path "rRecent ~> originalTop"
                                assert rep(N, oldS, i) == rRecent;

```

```
                                    assert ExFittingPath(N, G, rep(N,
                                        originalS, i), top(N, originalR), 1,
                                         part(N, oldS, top(N, oldR))); //
                                        precondition
                                    assert ExFittingPath(N, G, rep(N,
                                        originalS, i), top(N, originalR), 1,
                                         part(N, oldS, rRecent)); //
                                        substitute

                                    assert part(N, oldS, rRecent) <=
                                        newPartV;
                                    assert (\exists seq<int> P; 1 <= |P|;
                                            FittingPath(N, G, rep(N,
                                                originalS, i), top(N,
                                                originalR), P, part(N, oldS,
                                                 rRecent)) ==>
                                            FittingPath(N, G, rep(N,
                                                originalS, i), top(N,
                                                originalR), P, newPartV));

                                    assert ExFittingPath(N, G, rep(N,
                                        originalS, i), top(N, originalR), 1,
                                         newPartV);
                            }

                            assert ExFittingPath(N, G, rep(N, originalS, i),
                                top(N, originalR), 1, newPartV);
                    }
                    i++;
            }

            assert (\forall int x; ru(x, N) && rep(N, newS, x) == top(N,
                newR);
                    ExFittingPath(N, G, rep(N, originalS, x), top(N,
                        originalR), 1, part(N, newS, top(N, newR))));
}

requires AdjacencyMatrix(N, G);
requires UnionFind(N, oldS);
requires UnionFind(N, newS);
requires ru(x, N);
requires part(N, oldS, x) <= part(N, newS, x);
requires rep(N, oldS, x) == rep(N, newS, x);
requires CP(N, G, oldS, x);
ensures ExPathToRep(N, G, oldS, x) && ExPathFromRep(N, G, oldS, x);
ensures (\exists seq<int> P; 1 <= |P|;
        FittingPath(N, G, x, rep(N, oldS, x), P, part(N, oldS, x)) ==>
        FittingPath(N, G, x, rep(N, newS, x), P, part(N, newS, x)));
```

```
ensures (\exists seq<int> P; 1 <= |P|;
        FittingPath(N, G, rep(N, oldS, x), x, P, part(N, oldS, x)) ==>
        FittingPath(N, G, rep(N, newS, x), x, P, part(N, newS, x)));
ensures \result == CP(N, G, newS, x);
static pure boolean Lemma_CP_Maintained_Pure(int N, seq<seq<boolean>> G, seq<int> oldS, seq<int> newS, int x) = true;

requires AdjacencyMatrix(N, G) && UnionFind(N, oldS) && UnionFind(N, newS) && ru(x, N);
requires part(N, oldS, x) <= part(N, newS, x) && rep(N, oldS, x) == rep(N, newS, x);
requires ExPathFromRep(N, G, oldS, x);
ensures (\exists seq<int> P; 1 <= |P|;
        FittingPath(N, G, rep(N, oldS, x), x, P, part(N, oldS, x)) ==>
        FittingPath(N, G, rep(N, newS, x), x, P, part(N, newS, x)));
ensures ExPathFromRep(N, G, newS, x);
static pure boolean Lemma_PathFromRep_Maintained_Pure(int N, seq<seq<boolean>> G, seq<int> oldS, seq<int> newS, int x) = true;

requires AdjacencyMatrix(N, G);
requires UnionFind(N, oldS);
requires UnionFind(N, newS);
requires ru(start, N);
requires ru(end, N);
requires part(N, oldS, start) <= part(N, newS, start); //in reality they are always equal, but that is harder to prove.
requires ExFittingPath(N, G, start, end, 1, part(N, oldS, start) + {end});
ensures (\exists seq<int> P; 1 <= |P|;
        FittingPath(N, G, start, end, P, part(N, oldS, start) + {end}) ==>
        FittingPath(N, G, start, end, P, part(N, newS, start) + {end}));
ensures ExFittingPath(N, G, start, end, 1, part(N, newS, start) + {end});
static pure boolean Lemma_RootPathPart_Maintained_Pure(int N, seq<seq<boolean>> G, seq<int> oldS, seq<int> newS, int start, int end) = true;

// Proves that there exists a path in the graph from R[i] to top(N, R), fitting in CTotal.
//
requires AdjacencyMatrix(N, G) && Stack(N, R) && UnionFind(N, S) && ru(r, N);
requires |R| > 0 && RootPath2(N, G, R, S);
requires 0 <= i && i < |R|;
requires (\forall int j; 0 <= j && j < |R|; rep(N, S, R[j]) == R[j]);
requires R[i] == r; // && r == rep(N, S, r);
requires (\forall int j; i <= j && j < |R|; part(N, S, R[j]) <= CTotal);
```

```
ensures ExFittingPath(N, G, r, top(N, R), 1, CTotal);
static void Lemma_RootPathToTop(int N, seq<seq<boolean>> G, seq<int> R, seq<int> S, int r, int i, set<int> CTotal) {
        assert part(N, S, r) <= CTotal;
        assert r in CTotal;

        if (i == |R| - 1) {
                assert r == top(N, R);                                  // base case:
                assert FittingPath(N, G, r, r, [r], CTotal);
                assert ExFittingPath(N, G, r, r, 1, CTotal);
                assert ExFittingPath(N, G, r, top(N, R), 1, CTotal);          // top can always reach itself, path = [top(N, R)]
        } else {
                int nextRoot = R[i+1];                                  // inductive case:
                set<int> subset = part(N, S, r) + {nextRoot};
                assert ExFittingPath(N, G, r, nextRoot, 1, subset);           // head segment - from RootPath2 precondition

                Lemma_RootPathToTop(N, G, R, S, nextRoot, i+1, CTotal);         // recurse!
                assert ExFittingPath(N, G, nextRoot, top(N, R), 1, CTotal); // tail segments

                assert Lemma_ExFittingPath_Transitivity_Pure2(N, G, r, nextRoot, top(N, R), subset, CTotal, CTotal);    // concat head and tail segments
                assert ExFittingPath(N, G, r, top(N, R), 1, CTotal);          // conclusion!
        }

        assert ExFittingPath(N, G, r, top(N, R), 1, CTotal);
}

//'for all indices after the splitPoint':
requires AdjacencyMatrix(N, G) && Stack(N, R) && UnionFind(N, S) && ru(splitPoint, |R|);
requires |R| > 0 && RootPath2(N, G, R, S);
requires (\forall int j; 0 <= j && j < |R|; rep(N, S, R[j]) == R[j]);
requires (\forall int j; splitPoint <= j && j < |R|; part(N, S, R[j]) <= CTotal);
ensures (\forall int i; splitPoint <= i && i < |R|; ExFittingPath(N, G, R[i], top(N, R), 1, CTotal));
```

```
static void Lemma_RootPathToTop_All(int N, seq<seq<boolean>> G, seq<int>
     R, seq<int> S, set<int> CTotal, int splitPoint) {

         loop_invariant splitPoint <= idx && idx <= |R|;
         loop_invariant (\forall int k; splitPoint <= k && k < idx;
             ExFittingPath(N, G, R[k], top(N, R), 1, CTotal));
         for (int idx = splitPoint; idx < |R|; idx++) {
                 int r = R[idx];
                 Lemma_RootPathToTop(N, G, R, S, r, idx, CTotal);
         }

         assert (\forall int i; splitPoint <= i && i < |R|; ExFittingPath
             (N, G, R[i], top(N, R), 1, CTotal));
}
```