UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics and Computer Science

Verification of a model checking algorithm in VerCors

JOHANNES PETRUS HOLLANDER
student nr.: 1723081 e-mail: j.p.hollander@student.utwente.nl

Master's Thesis Computer Science
August 2021

Supervisors:
prof. dr. M. Huisman
Ö.F.O. Şakar, MSc
Committee Members:
prof. dr. M. Huisman
dr. C.E.W. Hesselman

Formal Methods and Tools research group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

ACKNOWLEDGEMENTS

The writing of this thesis was not an easy progress, but ultimately it was a fulfilling experience. From the first words read while researching the topic, to the final touches made just before the deadline, I would like to thank the following people who helped me write this thesis.

The completion of this thesis would not have been possible without the members of the committee and my supervisors: Marieke Huisman, Cristian Hesselman and Ömer Şakar. Their invaluable feedback and ideas have helped me a lot: especially Marieke's technical knowledge and guidance, Cristian's fresh perspective and of course the numerous fruitful meetings with Ömer, where we shared ideas, solved problems, and ultimately made this thesis happen.

Aditionally, I would like to thank Wytse Oortwijn, Mohsen Safari, Vincent Bloemen and Peter Lammich for their input for this thesis. Wytse an Mohsen shared their experiences with working with the VerCors tool set, Vincent helped me with the verification of the algorithm and Peter showed me his verification efforts using refinement techniques. There were also many other people within the VerCors community that were always willing to assist if I encountered a problem, and I very much appreciate their help.

Finally, thanks to all the others who helped me along the way to make this thesis happen!

ABSTRACT

Deductive software verification is a formal method to verify that the behaviour of a program satisfies a set of specifications. We are currently able to make use of highly automated techniques to verify complex programs, such as model checking algorithms. These model checking algorithms are high-level graph algorithms which are used to reason about the behaviour of software, by verifying properties of an abstract model of the software system. Because these model checking algorithms are becoming increasingly complex, the need for formal verification of the algorithms becomes more apparent. In this thesis we show how the VerCors tool set can be used to verify a sequential model checking algorithm, the set-based SCC algorithm. We then explore how the VerCors tool set can be improved for verifying these kinds of algorithms, both by reflecting on the verification in this thesis and by comparing different related verification techniques.

Keywords: VerCors, deductive verification, model checking, graph algorithms, automated verifiers, interactive provers, concurrent algorithms, strongly connected components, union-find

CONTENTS

1 INTRODUCTION

Deductive software verification is a formal method to verify that the behaviour of a program satisfies a set of specifications. This verification process is based on a system of logical inference. The field of deductive software verification has made significant progress since the idea was first put forward in the late 1960s. Where back then the deductive proofs for program correctness were handwritten, small and limited in scope, we are currently able to use highly automated techniques to verify complex programs in popular programming languages.

A contemporary and important application of the deductive program verification technique is the verification of *model checking algorithms*, which are a type of high-level graph algorithms that are employed by model checkers. Model checking is a method to reason about the behaviour of programs, making use of an abstract model of a software system. Model checkers automatically verify properties on this model, using the aforementioned model checking algorithms. These graph algorithms search the state space of the program, and try to find any behaviour that violates the specified property. An increase in software size and complexity has led to a combinatorial explosion of the state space, and the size of the models of these software systems has increased along with it. This in turn has led to researchers trying to find increasingly efficient model checking algorithms. Unfortunately, an increase in efficiency often comes with higher complexity of the algorithms as well.

Because compliance of a software system to its specification may be critical from both a functional and a safety perspective, it is important that we can be assured that if a model checking procedure is applied, it always gives the correct result: if there is a counter-example to be found, the model checking algorithm will always report it. This can only be achieved by proving the model checking algorithm correct. Proving correctness of algorithms employed by model checkers has traditionally been done manually, either on paper or with interactive provers. However, as the algorithms become more complex and employ advanced concepts to mitigate common problems (e.g. specific data structures or parallelism), these methods become more difficult to use. This is why the feasibility of mechanical verification, which has potential for re-use and automation, should be investigated.

1.1 Goal and research questions

Oortwijn [Oor19] already made a start in this investigative direction by automatically verifying a parallel version of the NDFS algorithm [LLvdP^{+}11] in the VerCors tool set [BDHO17], using deductive verification. This work has laid the groundwork for this thesis. The goal of this report is to identify and provide possible improvements for the VerCors tool set so this kind of verification can be carried out more efficiently in the future.

This thesis endeavours to answer the following two research questions in order to further the work on this subject:

RQ1. *What techniques are involved with proving the correctness of a high-level model checking algorithm using the VerCors tool set?*

RQ2. *How can the VerCors tool set be improved for the verification of other (parallel) graph algorithms?*

   (a) *How suitable is VerCors for the verification of the model checking algorithm from a user perspective?*

   (b) *What can be learned from other verification techniques and tools to improve the verification of graph algorithms with VerCors?*

1.2 Approach

Oortwijn describes the verification of the correctness of a parallel nested depth-first search (PNDFS) algorithm. This is, as far as we know, the first time a mechanical proof of such an algorithm has been done. The tool set that was used is VerCors, which is being developed and maintained by the Formal Methods and Tools research group at the University of Twente. This algorithm is one of the many algorithms that can be used for reachability analysis, and detection of accepting cycles and strongly connected components (SCC's) in model checking [BBDL$^+$18]. The verification of PNDFS is just the first step in using VerCors for the verification of graph algorithms. The method for doing this kind of verification is therefore still somewhat experimental and not very defined and generalised. In this thesis we find several points of improvement for the tool set.

To discover any of these improvements, we try to understand the methods of program verification in VerCors by verifying at least one other model checking algorithm. Oortwijn recommends a pair of algorithms that partition a given graph in strongly connected components [Blo19], which are already conceptualised, (manually) proven and implemented. This means that the desired completeness/soundness properties are already formalised and can be transformed into a VerCors verification. While these two algorithms are strong contenders, we also investigate other algorithms that have potential to be useful. We use the verification of this algorithm, along with the verification of PNDFS and other previous efforts using different verification techniques, to help answer the two research questions.

1.3 Contributions

This thesis contributes both to the practical and theoretical side of the deductive verification of graph algorithms in VerCors. This thesis has two main contributions, corresponding to the two research questions listed earlier in this chapter.

The first contribution is the (partial) verification of a sequential model checking algorithm, the set-based SCC algorithm, answering RQ1. This verification is carried out with the VerCors tool set, and it makes a start with proving the soundness and correctness of the algorithm. The verification is based on a proof outline of the algorithm [Blo19], and proves the two most important invariants that are needed for the complete proof. This verification is a continuation of the effort started in Oortwijn's thesis [Oor19].

The second main contribution answers RQ2 by reviewing the verification of the set-based SCC algorithm and exploring different related verification techniques. We look at the verification in this thesis from the perspective of a VerCors user, more specifically at reusability, feature support, and user experience. The related verification techniques are compared to VerCors. Finally, based on these two analyses, we suggest a set of improvements to VerCors, and give an idea of how these improvements could be realised.

1.4 Thesis structure

This thesis is organised as follows:

Chapter 2 and chapter 3 are background chapters on model checking and deductive verification, respectively. Chapter 4 answers RQ1 and chapter 5 uses these findings to answer RQ2. The recommended reading order is in-order, though the two background chapters can be read independently of each other. Below a brief description of the contents of each chapter is provided.

Chapter 2 provides the necessary background information to understand the problem that (LTL) model checking algorithms try to solve: the emptiness-check problem. Furthermore, it contains information about the challenges of model checking, systems and specifications, graph searches, and automata. General concepts about model checking algorithms, such as search orders, accepting cycles and strongly connected components, and parallelism are alse explored.

Chapter 3 gives information about deductive software verification, and the VerCors tool set. Hoare logic, wp-reasoning and separation logic - the core concepts behind deductive verification - are introduced and the architecture and methodology of the VerCors tool set is explained.

Chapter 4 introduces a model checking algorithm, more specifically the set-based SCC algorithm. It lays out the several algorithm specific concepts, and it provides a pseudocode representation of the algorithm along with an example run. Besides this, the chapter gives the correctness criteria and proof outline for the algorithm, before going through the complete verification effort that is carried out using VerCors.

Chapter 5 analyses the verification of the set-based SCC algorithm from the perspective of a VerCors user, looking at usability and user experience. It then explores related efforts in the fields of verification using both interactive and automated theorem provers, before finally suggesting improvements to the VerCors tool set.

2 LTL MODEL CHECKING

As mentioned in the introduction to this report, model checking is an important application of deductive verification. This chapter provides enough background information to the reader to be able to understand the problem model that checking algorithms try to solve, as well as how these algorithms work. We will be focusing on automata-theoretic model checking, more specifically LTL model checking (a symbolic method of model checking), and when when using "model checking" in the remainder of this report we will be referring to this specific type.

2.1 Background of LTL model checking

The algorithms this project is concerned with all have a similar goal, namely to check if a model of a system conforms to the provided specification - a process called model checking. To understand how these algorithms work some preliminary knowledge is required. In this section this information is presented, and eventually the emptiness-check problem is defined, along with two common properties of automata that can be used to solve it: accepting cycles and SCCs. Solving the emptiness-check problem is a method that can be used for model checking. For this reason, solving this specific problem is often the goal of the aforementioned model checking algorithms.

2.1.1 Model checking and its challenges

LTL model checking is the practice of taking a system description and a system specification, transforming them into some type of state-transition graph (the model) and temporal-logic formula (the specification) respectively, and finally checking whether or not the model satisfies the specification. The specification describes all properties that the software system (and by extension the model of that system) should have. Two common types of properties that the specification expresses are *safety* and *liveness.* In short, safety properties are properties that specify that *something bad never happens*, while liveness properties ensure that *something good eventually happens.*

Model checking algorithms for safety properties are relatively straightforward (at least compared to its liveness counterparts): they check if a certain erroneous state can be reached - if so, then the system is not safe. An example could be a traffic light system where we want to ensure we can never reach a state where the red, yellow and green lights are all on at the same time. Checking liveness properties is more complicated, since it involves analysing infinite running systems. In our example of a traffic light, a liveness property could be that each light will always turn green at some point in the future.

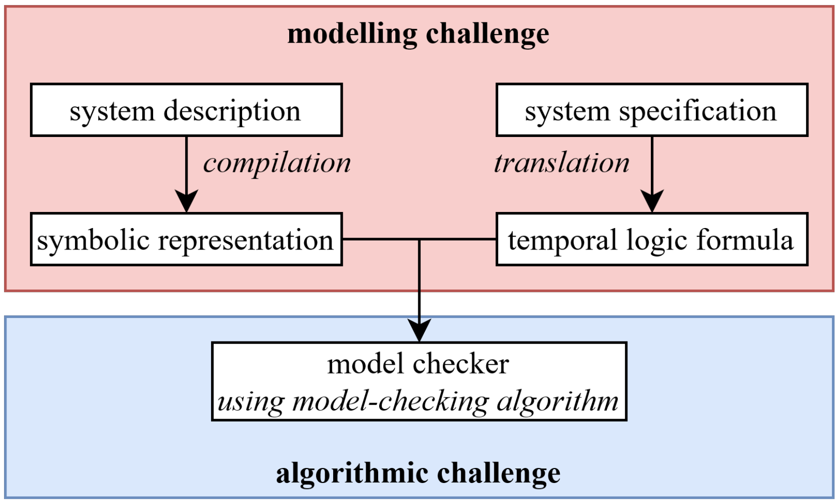


Figure 1: *Basic diagram of the model checking methodology. [CHVB18]*

In this research we focus on methods for verification of liveness properties. An example of such a method involves the emptiness-check problem, further explained in section 2.1.5. This method is a way of finding a counterexample to a liveness property.

There are two main challenges that arise during the process of defining a model checking method [CHVB18]:

- **The modelling challenge:** How can we best represent a system using a model that is both expressive and efficient? It is important that none of the critical characteristics of the system get lost in modelling process, because this could render the model checking outcome useless since it might not apply to the system. On the other hand, we want to minimise the size and complexity of the model as much as possible to make the model checking more efficient. Expressiveness often comes at the cost of efficiency, and vice-versa, so finding a balance between these two aspects is important.
- **The algorithmic challenge:** How can we design model checking algorithms that scales well and can be used to solve real-life problems? In general, real-life systems are very large and complex, and so the models of these systems will be too. This means that the model checking algorithms should be able to run on very large models in order for them to be useful beyond a small scale academic setting.

If we look at a basic diagram of the model checking methodology, shown in figure 1, we can see at which phase of the model checking process these two challenges arise. In sections 2.1.2 and 2.1.3 the difficulties with these challenges will be explored further. While this paper is mainly concerned with the verification of a model checking algorithm (and thus the algorithmic challenge), to understand the goal and design of the algorithms we need knowledge of the possible solutions to the modelling challenge as well.

2.1.2 Systems and specifications

The first step in the model checking procedure is to compile the system description into a model which has a form that can be checked by algorithms. Due to the nature and size of most modern programs it is often not feasible to use a *structural* method to generate this model. These methods use the syntactic expression of the system, so the code itself, to construct the model. However, it generates such a large state space that the limiting factor is almost always memory space [CHVB18]. Instead, we often favour *symbolic* methods where states and transitions are not explicitly enumerated, but rather expressed in a symbolic logic (for example binary decision diagrams or propositional formulas). Using this symbolic encoding can greatly reduce the state space and so improve performance of the verification. An obvious downside to employing symbolic methods is that the compilation of the model from the system description is less trivial, and we need to take into account the greater abstraction layer the model introduces and its potential to lose crucial information in the abstraction process.

The actual structure used in practice depends on the demands of the model and which properties of the system need to be expressed, but most symbolic model checking methods use Kripke structures. Kripke structures are a form of automata and, along with an encoding of the system specification in a temporal logic, make up the basis for the model checking procedure. In this research we use the basic form of Kripke structures and linear-time temporal logic (LTL). In the remaining part of this section these two concepts will be explained in further detail.

Kripke structures

Kripke structures are a generalised form of automata, more specifically finite directed graphs where we label vertices with atomic propositions. We use the terminology of "states" and "transitions" for vertices and edges, respectively. The property of Kripke structures essential for model checking is that each state is labelled with an assignment encoding the state of the system, allowing for the aforementioned expression in symbolic logic (specifically using logical propositions, for example LTL).

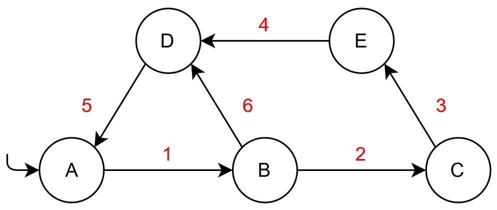
An *assignment* is a function $x : \mathrm{AP} \to \mathbb{B}$, where $\mathbb{B} = \{\top, \bot\}$ is the set of boolean values, and AP is a finite set of atomic propositions. *Atomic (logical) propositions* are statements that are true or false, and cannot be divided into smaller propositions, and an assignment assigns a truth value to each of the propositions in AP. Using this definition of an assignment, Kripke structure can be represented as a tuple [BBDL+18] $K = (Q, \iota, \delta, \ell)$ where:

- Q is a finite set of states,
- $\iota \in Q$ is the initial state,
- $\delta \subseteq Q \times Q$ is a set of transitions,
- $\ell : Q \to \mathbb{B}^{\mathrm{AP}}$ is a function labelling each state with an assignment.

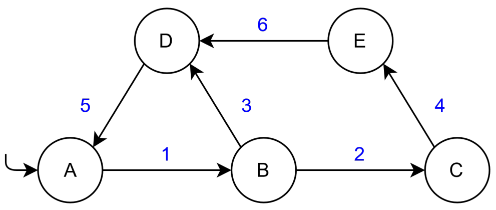
Linear-time temporal logic

Linear-time temporal logic (LTL) is a grammar for logic formulas that expresses some property of a system model (in model checking this property is defined in the specification). As the name of the grammar suggests, these properties can contain a temporal element. For example, the property "ϕ eventually holds", with ϕ any LTL formula can be expressed as $\mathsf{F}\phi$. LTL uses the atomic propositions in AP to construct formulas ϕ using the following grammar:

$$\phi ::= \top \mid \bot \mid a \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \,\mathsf{U}\, \phi \mid \phi \,\mathsf{R}\, \phi \mid \mathsf{F}\,\phi \mid \mathsf{G}\,\phi \mid \mathsf{X}\,\phi$$



(a) *DFS*



(b) *BFS*

Figure 2: *An example of the DFS and BFS search orders, starting from state A.*

Besides the logical operators and a single atomic proposition a, we can express the following:

- $\psi \,\mathsf{U}\, \phi$ - **U**ntil: ψ holds at least until ϕ becomes true (which *will* happen at some point).
- $\psi \,\mathsf{R}\, \phi$ - **R**elease: ϕ holds until and including when ψ becomes true, and if ψ never becomes true ϕ always holds (so ψ "releases" ϕ).
- $\mathsf{F}\, \phi$ - **F**inally: ϕ will eventually hold.
- $\mathsf{G}\, \phi$ - **G**lobally: ϕ holds everywhere.
- $\mathsf{X}\, \phi$ - Ne**x**t: ϕ will hold in the next state

LTL sometimes also includes the operators W and M, which stand for a weak until and a strong release respectively. The meaning of W is similar to U, but it is not needed for ϕ to become true at some point in the future. M is similar to R, only here ψ needs to become true eventually.

2.1.3 Graph searches

In section 2.1.2 we established the use of finite directed graphs (graphs consisting of vertices connected by directed edges) as an expression of the state space. This means that we could use *graph traversal algorithms* working on directed graphs to solve the algorithmic challenge from 2.1.1. These algorithms are well-studied and widely used in many different areas where the goal is to iterate over the vertices of a graph, and so too in the domain of model checking. While this gives us promising candidates for algorithms that can solve the algorithmic challenge, we need to make sure that the algorithms can be used on very large graphs. In the case of model checking, one solution for this is to use *on-the-fly graph exploration.* In the remainder of this section the concepts behind the basic graph traversal methods and on-the-fly exploration are explained.

Graph traversal algorithms

Common types of problems that can be solved by graph traversal algorithms are checking the reachability of a vertex, testing the planarity of a graph, finding the shortest path between two vertices or finding certain structures such as cycles. Two basic methods these algorithms can and often do employ are *depth-first search* (DFS) and *breadth-first search* (BFS). Both DFS and BFS are orders in which a directed graph can be traversed, but the difference lies in which vertexes are processed first. With DFS, children of a vertex are explored first, and once all children are explored we backtrack and explore the sibling vertices. Using BFS, we do the opposite, where the siblings are visited before processing the child vertices. The difference between the two orders is illustrated in figure 2. In this figure the edges between vertices are annotated with a number, red for DFS in figure 2a and blue for BFS in figure 2b. This number indicates the traversal order of both methods. In this example we can clearly see the difference if we look at which edge is visited third: for DFS this is the successor of vertex C, while for BFS this is the sibling of vertex B.

Which of the two orders an algorithm uses depends on the requirements the algorithms have, both in terms of functionality and efficiency. For example, some algorithms rely on the DFS search order for their correctness, while others are easily parallelisable if BFS is used.

On-the-fly graph exploration

It is not always feasible to store a complete graph in memory due to the fact that the size of the graph may be too large. In this case a solution could be to use implicit graphs instead, and the exploration becomes *on-the-fly*: the graph is generated as the algorithm traverses it. Where for explicit graph representations all vertices and edges are known and stored in memory, an implicit graph uses a function that returns the successors of a given vertex. This so-called *next-state function*, paired with a known initial state, can represent a graph without initial knowledge of the rest of the graph. An example of how implicit graphs can be used is with finding solutions for a Rubix cube puzzle. It is not too difficult to calculate all successor states if the current state is known (we know which moves can be made), and since the state space of a Rubix cube is extremely large it cannot feasibly be stored in memory.

Besides potentially reducing the amount of memory needed to store the graph, it is also important that both DFS and BFS can be used with an implicit graph. Both these methods only need to know the successors of the current vertex, and it's ancestors (which can be guaranteed by keeping a search stack). The searches do not need to be aware of the complete graph.

It is possible to use implicit models in the model checking process [BBDL$^+$18], allowing for on-the-fly model checking algorithms. Using this method, instead of storing the complete state space of the combination of the model and a specification we make an intermediate *implicit product automaton*. We then use this implicit product automaton to generate the final automaton used by the model checking algorithms on-the-fly. This generated automaton would by definition be no different than the explicit one, and is equivalent to the (explicit) *synchronised product* described in 2.1.4.

2.1.4 Büchi automata

As discussed in section 2.1.2, the state space of the model of a system is usually expressed as a Kripke structure. In order to combine this structure with a property we want to check (an LTL formula) we need to introduce a new type of automaton, the *Büchi automaton*. Any LTL formula can be transformed into an Büchi automaton. This automaton can be combined with a Kripke structure to form a new Büchi automaton, which can then be used by the model checking algorithms.

There are four types of Büchi automata, based on two options. The first option is for the automata to have transition-based or state-based acceptance, and the second option is to have classical or generalised Büchi acceptance. All of these types of automata can express the same languages but can be more or less efficient for different purposes and the emptiness-check problem can be solved in different ways. This choice has an impact on efficiency and complexity of the model checking algorithms, but all types can be transformed into any other type by simple transformations - though often at the cost of increasing the size of the automata (in terms of states and transitions). In the rest of this section the four types are explained in more detail, as well as the use of Büchi automata in the transformation of LTL formulas and the synchronised product with a Kripke structure.

TGBA

A TGBA is a *Transition-based Generalised Büchi Automaton.* This means that transitions can be accepting. It is represented as a tuple (in [BBDL+18] $A = (Q, \iota, \delta, n, M)$ where:

- Q is a finite set of states,
- $\iota \in Q$ is the initial state,
- $\delta \subseteq Q \times \mathbb{B}^{\mathrm{AP}} \times Q$ is a set of transitions,
- n is an integer specifying the number of acceptance marks,
- $M : \delta \to 2^{[n]}$ is a marking function that specifies a subset of marks associated with each transition.

For these kinds of automata, accepting runs are those runs that take at least one transition for every acceptance marks *infinitely often.* The set of all accepting runs of automaton A is denoted as $\mathscr{L}(A)$.

TGBAs have the property that they can be transformed, or "degeneralised" into an equivalent SBA with mat most $(n+1)|Q|$ states, or into a TBA with $n \cdot |Q|$ states. All LTL formulas can also be transformed into TGBAS with at most $2^{|\phi|}$ states and $|\phi|$ acceptance marks (which is n). There exist methods and tools for converting TGBAs into SBAs/TBAs, and for converting LTL formulas into TGBAs, such as ltl2ba[1].

SGBA

An SGBA (*State-based Generalised Büchi Automaton*) is very similar to a TGBA, with the same definitions for Q, ι, δ and n, but instead of having accepting marks on the transitions it has them on the states, so $M : Q \to 2^{[n]}$. Accepting runs are those runs that pass trough at least one state for every acceptance mark infinitely often. Similarly to TGBAs, an SGBA can also be transformed into all other types of Büchi automata.

SBA and TBA

SBAs and TBAs (*State-based Büchi Automata* and *Transition-based Büchi Automata* are just SGBAs and TGBAs but with $n = 1$, so only one accepting mark. This means that all operations that can be carried out on SGBAs and TGBAs can also be applied to SBAs and TBAs, such as taking the synchronised product with a Kripke structure.

[1]Main page for LTL 2 BA: `http://www.lsv.fr/~gastin/ltl2ba/` (accessed 24-06-2021)

LTL to Büchi automaton

LTL formulas can be transformed into TGBAs, and they can in turn be transformed into SBAs/TBAs, so it follows that LTL formulas can be transformed into SBAs. This transformation can result in an SBA with a potentially exponential size w.r.t. the size of the LTL formula. However, this upper limit is almost never reached in practice [BBDL+18], making the transformation a valid strategy to employ.

For model checking, the negation of the LTL formula expressing the property we want to check is used. This is done to show the presence of a counterexample: if we find a run satisfying the negation of a property, we know that the same run does not satisfy the property itself.

Synchronised product

The way that the Kripke structure representing the model state space and the Büchi automaton of the negation of the LTL formula are combined is using the synchronised product. The result of this operation is another Büchi automaton of the same type as the input Büchi automaton. Here we give an example using a TGBA, but the process for all other forms of Büchi automaton is similar.

For a Kripke structure $K = (Q_1, \iota_1, \delta_1, \ell)$ and an TGBA $A = (Q_2, \iota_2, \delta_2, n, M)$ the synchronised product is a TGBA $K \otimes A = (Q', \iota', \delta', n, M')$ where:

- $Q' = Q_1 \times Q_2$,
- $\iota' = (\iota_1, \iota_2)$,
- $((s_1, s_2), x, (d_1, d_2)) \in \delta' \iff (s_1, d_1) \in \delta_1 \wedge \ell(s_1) = x \wedge (s_2, x, d_2) \in \delta_2$,
- $M'(((s_1, s_2), x, (d_1, d_2))) = M((s_2, x, d_2))$.

This new TGBA has the property that $\mathscr{L}(K \otimes A) = \mathscr{L}(K) \cap \mathscr{L}(A)$. When using an SGBA the only difference is that $M'(s_1, s_2) = M(s_2)$. The size of the new automaton is the product of the input Kripke structure and Büchi automaton, so $|Q'| = |Q_1| \cdot |Q_2|$, which is obviously not ideal if we want to limit the size of our automaton. However, the conversion of an LTL formula to an automaton often produces a small Büchi automaton, and so this impact remains manageable. Furthermore, the states in Q' that are reachable from ι' can be substantially smaller than the complete set of states, and only these states need to be explored by the algorithm. This last fact is especially relevant if we use implicit representations, as discussed in 2.1.3.

2.1.5 Emptiness-check problem

The goal of many model checking algorithms is to solve the emptiness-check problem for an automaton B, which is the synchronised product of the model and the specification as described in section 2.1.4. Essentially the emptiness-check problem is the question whether $\mathscr{L}(B) = \emptyset$ (the language of the automaton B, so the set of all infinite accepting runs) is empty. This can be done on all types of Büchi automata, but TGBA and SBA are most commonly used. Two ways to disprove the existence of such an accepting run are finding an accepting cycle reachable from the initial state or dividing the graph into strongly connected components (SCCs) and checking their acceptance and reachability. Both of these concepts will briefly be explained later in this section.

The definitions of accepting cycles and strongly connected components can be used to define equivalence between statements about how to check for emptiness. These equivalences are stated in Theorem 1, from [BBDL+18]:

Theorem 1 (Emptiness-check problem)**.** *Let ϕ be an LTL formula, $A_{\neg\phi}$ an automaton with n acceptance marks such that $\mathscr{L}(\neg\phi) = A_{\neg\phi}$, and K a Kripke structure. The following statements are equivalent:*

1. $\mathscr{L}(K) \subseteq \mathscr{L}(\phi)$,
2. $\mathscr{L}(K) \cap \mathscr{L}(A_{\neg\phi}) = \emptyset$,
3. $\mathscr{L}(K \otimes A_{\neg\phi}) = \emptyset$,
4. *$K \otimes A_{\neg\phi}$ has no reachable, accepting cycle; or in case $n \leq 1$ no reachable accepting elementary cycle,*
5. *$K \otimes A_{\neg\phi}$ has no reachable, accepting SCC.*

Further explanation of Theorem 1: Point 1 in the theorem can be read as: *the language of K is a subset of the language of the LTL formula ϕ we want to check, i.e. ϕ holds for all runs in K*. Point 2 can be read as: *the intersection of K and the negation of ϕ is empty, i.e. there are no runs in K where ϕ does not hold.* Point 3 is very similar to point 2 but with K and $A_{\neg\phi}$ combined into one automaton per the $\otimes$ operator (the synchronised product), and points 4 and 5 can be shown (by the definition of $\mathscr{L}$) to have the same meaning as point 3 but expressed using accepting cycles and SCCs, respectively.

This theorem shows how accepting cycles and strongly connected components can be used to verify some property on a model using points 4 and 5. Most well-known algorithms employ one of these two strategies (see appendix A for a list of algorithms). Below the definitions for cycles and different types of strongly connected components are given.

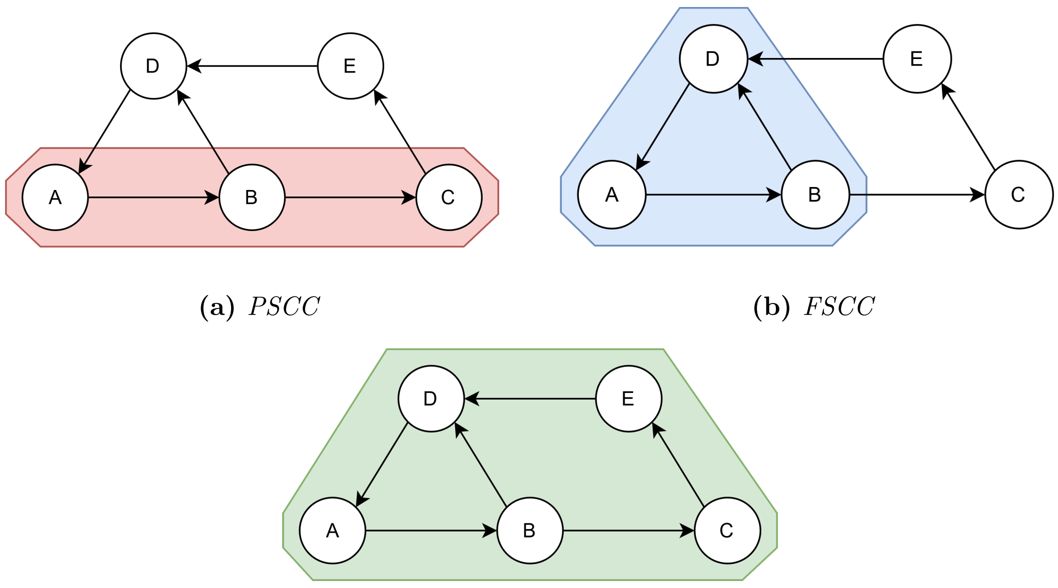
Following the notation used in [Blo19], given some directed graph $G = (V, E)$ where V is the set of states and E the set of transitions, we have the definition for paths as given below.

Definition 1 (Path)**.** We denote a transition $(v, w) \in E$ as $v \to w$. A path with length k is defined as a sequence of states $\langle v_0, \dots, v_{k-1}\rangle$ where $\forall i \in [0 \mathrel{..} k-1] : v_i \in V$ and $\forall i \in [0 \mathrel{..} k-1) : v_i \to v_{i+1}$. A path from v to w, $\langle v, \dots, w\rangle \in V^*$, is denoted as $v \to^* w$. If there exists a path from v to w and from w to v (so $v \to^* w \wedge w \to^* v$) then the two states are strongly connected, and we denote this as $v \leftrightarrow w$.

Accepting cycles

To check for emptiness using cycles, we have to check if an (Büchi) automaton B (which is the synchronised product of some Kripke structure K derived from some model M and automaton $A_{\neg\phi}$ for some temporal property ϕ) contains an *accepting cycle* reachable from the initial state to reject the emptiness property, and if the number of accepting marks is one or zero, we have to check for an *elementary* accepting cycle, also reachable from the initial state. Intuitively, if such a cycle exists and it is reachable from the initial state then there is an infinite accepting run in automaton $B = K \otimes A_{\neg\phi}$, so $\mathscr{L}(B) \neq \emptyset$. Referring to Theorem 1 point 3 we can see that the emptiness check failed.

The definitions of cycles, accepting cycles and elementary cycles are given below.



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

Figure 3: *The three types of SCC shown for a simple graph [Blo19].*

Definition 2 (Cycle)**.** Given a path $c \in V^*$ of length k, where $c = \langle v_0 \dots v_{k-1} \rangle$, then c is a cycle iff $v_{k-1} \to v_0$. A cycle is an *elementary* cycle iff it goes through k different states, i.e. $\forall i \in [0 \mathrel{..} k-1) : \forall j \in (i \mathrel{..} k-1] : v_i \neq v_j$. A cycle is *accepting* for a TGBA iff for some number of acceptance marks n and a marking function M, $\forall i \in [1 \mathrel{..} n] : \exists j \in [0 \mathrel{..} k-1] : n \in M(v_j \to v_{j+1 \bmod k})$ and for an SGBA iff $\forall i \in [1 \mathrel{..} n] : \exists j \in [0 \mathrel{..} k-1] : n \in M(v_j)$.

Strongly connected components

Another way to check for emptiness, is to check for *accepting strongly connected components*. If we divide the graph into SCCs, and then check if there exists an accepting SCC reachable from the initial state we can check for the emptiness property. A property of an accepting SCC is that it always contains an accepting cycle (implied by Definition 5), and so we can again use Theorem 1 to prove or disprove the existence of accepting runs.

In [Blo19] three types of SCC are described: a partial SCC (PSCC), a fitting SCC (FSCC) and a proper SCC. Below the definitions of these types are given, where each type of SCC builds on the previously defined types, and in figure 3 examples of the three types of SCC are shown.

Definition 3 (PSCC)**.** A PSCC is a set of states in a graph, such that for all pairs of sets in those states there exist paths to each other. *These paths can include states and transitions outside of the PSCC.* This means that a *non-empty* state-set $C \subseteq V$ is an PSCC iff $\forall v, w \in C : v \leftrightarrow w$

Definition 4 (FSCC)**.** An FSCC is a PSCC where the paths between the pairs of states do not include states and transitions outside of the FSCC. So, a *non-empty* state-set $C \subseteq V$ is an FSCC iff $\forall v, w \in C : \exists \langle v, v_0 \dots v_n, w \rangle \in V^* : \forall k \in [0 \mathrel{..} n] : v_k \in C$

Definition 5 (SCC)**.** An SCC is a *maximal* FSCC, i.e. FSCC $C \subseteq V$ is an SCC iff there does not exist an FSCC $C' \subseteq V$ such that $C \subset C'$. An SCC is *trivial* iff it consists of a single state with no self-loop, so $|C| = 1$ with $C = [v]$ and $v \nrightarrow v$. For an TGBA a non-trivial SCC is accepting for some number of acceptance marks n and a marking function M if the transitions induced by the SCC cover all acceptance marks, i.e. $\forall i \in [1 \mathinner{..} n] : \exists v, w \in C : n \in M(v \to w)$, and for an SGBA a non-trivial SCC is accepting if the states in the SCC cover all acceptance marks, i.e. $\forall i \in [1 \mathinner{..} n] : \exists v \in C : n \in M(v)$.

2.2 Algorithms

Several algorithms can be used to solve the emptiness-check problem. These algorithms can be broadly categorised by three variables: search order (BFS or DFS), search goal (accepting cycles or SCCs), and parallelism. Each of these factors influences the performance, complexity and application of the algorithms in different ways. Furthermore, all algorithms can be used as-is or on-the-fly, depending on what is needed. In the remainder of this section we will explain the impact of these three choices, and in appendix A we list the most used model checking algorithms along with a brief description. In chapter 4 of this report we will explore one specific sequential algorithm, based on finding SCCs using the DFS search order.

2.2.1 Search order

While there are other search orders than DFS and BFS, these two are almost always used, because they do not require the entire graph to be known in advance and so on-the-fly algorithms (as discussed in section 2.1.3) can also be used [Blo19]. The choice then - between DFS and BFS - depends on the requirements of the algorithm.

Most sequential emptiness-check algorithms use DFS as their search order. This is because by its nature, cycles can be easily detected using DFS: if in our search we encounter a successor-state that has been encountered before in the search, we know there is a cycle since that successor-state must also be an ancestor-state. As mentioned before we only ever need to compute the successors of a state and never the predecessors, so on-the-fly processing can be used.

Opposed to DFS, BFS-based exploration algorithms can not easily detect cycles, and require some additional bookkeeping to manage this. However, there are other properties of BFS that are very useful, most notably the higher potential for parallelisation when compared to DFS [BBDL$^+$18], which is explained in section 2.2.3.

2.2.2 Accepting cycles and SCCs

The choice between algorithms that detect accepting cycles and those that decompose graphs into SCCs depends on the requirements of the model checking. For example, two well known algorithms are *nested DFS* (NDFS, for finding accepting cycles) and algorithms based on the classic Tarjan's algorithm (for decomposing into SCCs). NDFS is generally more space efficient, but SCC-based algorithms can produce more concise counterexamples that can be analysed better [VBBB09]. So if memory constrains could be an issue, using NDFS is more practical, but if it is critical that the counterexamples can be understood SCC-based algorithms might be more useful.

2.2.3 Parallelism

With modern computers that can have multiple processors and cores we often want to make use of all available resources. This can be done through parellelisation, and in the case of model checking this means (re)designing algorithms [Blo19]. As mentioned before, BFS-based algorithms lend themselves well to parallelisation: the exploration of each state can be delegated to a job for each successor, and these jobs can be executed in parallel. With DFS it is more difficult, since there is no obvious way to divide the work. Both search orders however almost always need additional bookkeeping and so trade time efficiency for memory utilisation.

So the trade-off with a (successful) parallelisation of an algorithm or method is that it is bound to be more complex, and so also more error-prone. Chapter 3 provides a method for ensuring the correctness of these algorithms.

3 DEDUCTIVE SOFTWARE VERIFICATION

In chapter 4 we use the VerCors tool set employing *deductive software verification techniques* to reason about the verification of a model checking algorithm. This type of software verification relies on the principles of Hoare logic and, in our case, Concurrent Separation Logic, which will both be explained in section 3.1. The VerCors tool set is used to carry out the verification in chapter 4 and we lay out the architecture, background and methodology of the tool set in section 3.2.

3.1 Background of deductive verification

Deductive software verification is the practice of reasoning logically about a program. It does this by starting with basic axioms and premises about small program statements, and it then uses these basic premises to derive greater logical conclusions about a program. To illustrate this, we start with an informal example, before formalising the verification process using *Hoare logic* [Flo67, Hoa69] and *Concurrent Separation Logic* (CSL) [O'H07, Bro07]. Both Hoare logic and CSL are compositional, i.e. proofs about smaller programs can be used to verify larger programs that are composed of those smaller programs. Furthermore, both logics consist of syntactic proof rules, meaning that they look at the syntax of programs as opposed to its semantics.

Informal example

In the code-snippet below we show a simple swap program, consisting of an assumption that initially x and y are not equal, the swap code, and finally an assertion. The goal of this example is to show how deductive techniques can be used to make sure the assertion does not fail. In other words, we want to verify that in the program state in line 5 the values of x and y are not equal. N.B.: we assume the assignments are pass-by-value, so $a := b$ means that a now references the *value* of b.

```
1 assume $\neg(x = y)$
2 $tmp := x$
3 $x := y$
4 $y := tmp$
5 assert $\neg(x = y)$
```

Intuitively, the assert statement could be proven as follows:

1. At the beginning of the program, we know nothing about the state of the program except that the values of x and y are not equal, which is assumed in line 1.
   knowledge: $\neg(x = y)$
2. After line 2 we know that the value of *tmp* is the same as the value of x. This is based on the premise that the assignment operator := changes the state of the program in this specific way. By extension, we can also logically conclude that $\neg(tmp = y)$.
   knowledge: $\neg(x = y),\ tmp = x,\ \neg(tmp = y)$
3. In line 3 we change the value of y to the value of x. Using the same premise as in step 2, we now know that x has the same value as y, and they are both different from *tmp*.
   knowledge: $x = y,\ \neg(tmp = x),\ \neg(tmp = y)$
4. Line 4 assigns *tmp* to y, and we change our knowledge again using the same premise as before.
   knowledge: $\neg(x = y),\ \neg(tmp = x),\ tmp = y$
5. Finally, we check the assertion in line 5 and find that it passes, since we know $\neg(x = y)$ is true, as shown in the previous step.

3.1.1 Formalisation of deductive verification

While the informal example we previously looked at illustrates the idea of deductive verification, we would like a formalisation of these premises and axioms we used. In this section such formalisations are introduced: Hoare logic and weakest-precondition reasoning.

Hoare logic

Hoare logic (or sometimes Floyd-Hoare logic) [Flo67, Hoa69] is such a formalisation and often regarded as one of the foundations of deductive reasoning about software. Using this logic, we can reason about the correctness of sequential imperative programs.

Essential to Hoare logic are so-called *Hoare triples.* These triples consist of a *precondition* $\mathcal{P}$, *postcondition* $\mathcal{Q}$ and a program C. The notation for such a triple is as follows: $\{\mathcal{P}\}C\{\mathcal{Q}\}$. $\mathcal{P}$ and $\mathcal{Q}$ are logical assertions, i.e. assertions about the program state, and are usually expressed in first-order logic. The logical assertions that form the precondition are *assumed* to be satisfied before the execution of the program, while the assertions in the postcondition are satisfied afterwards. So intuitively, a Hoare triple can be read as follows: starting from a program state satisfying the precondition $\mathcal{P}$, after executing the program C the program state will satisfy the postcondition $\mathcal{Q}$. In the context of deductive verification, for a program C to be *partially correct* it means that given $\mathcal{P}$, *if* C terminates, then $\mathcal{Q}$ holds. To prove the *total correctness* of C, we need to specify an additional proof that the program always terminates. The notation of a Hoare triple that requires total correctness is often written $[\mathcal{P}]C[\mathcal{Q}]$.

By combining basic inference rules from Hoare logic we can compose proofs for larger programs. These proofs show that programs comply to the given specifications, namely the pre- and post-conditions $\mathcal{P}$ and $\mathcal{Q}$. If such a proof can be composed for a Hoare triple $\{\mathcal{P}\}C\{\mathcal{Q}\}$, we say that the program is verified and use the notation $\vdash \{\mathcal{P}\}C\{\mathcal{Q}\}$.

The core inference rules for Hoare logic cover **skip**, **assignment**, **sequential composition**, **conditionals**, **loops** and **consequence**. Using these rules, we can verify simple programs with assignments, if-then-else constructs, and (while-)loops. Furthermore, the skip rule covers the case of empty statements, for example in the else part of an if-then-else construct. The sequential composition rule ensures we can split the program into two sub-programs, and the consequence rule allows for strengthening the precondition and weakening the postcondition.

The precise definitions of all rules can be found in the original work of Hoare [Hoa69]. Hoare uses a slightly different notation where a Hoare triple is written $\mathcal{P}\{C\}\mathcal{Q}$ instead of $\{\mathcal{P}\}C\{\mathcal{Q}\}$, but the rules are the same. To get an idea of how Hoare logic is used we will discuss two of these rules needed to show how to derive a proof for our previous informal example. The rules are written using the notation of natural deduction, as axiom schemas or inference rules. The next part of this section describes two Hoare rules: the axiom for assignments, and the inference rule for sequential composition:

HT-ASSIGN Assignments of the form $x := e$ are handled by this axiom. Assign statements assign the value of the expression e to the variable x, thus changing the program state. Logically, some postcondition $\mathcal{Q}$ is true after an assignment if it held before the assignment as well, but with all free occurrences of x substituted by e. We denote this as $\mathcal{Q}[x/e]$.

$$\frac{}{\vdash \{\mathcal{Q}[x/e]\}x := e\{\mathcal{Q}\}}\ \textsc{ht-assign}$$

HT-SEQ Sequential composition of two programs is handled by this inference rule, and it is one of the most important rules of the set since it provides the compositionality of Hoare logic. If two programs C_1 and C_2 are composed sequentially as $C_1; C_2$, and have a precondition $\mathcal{P}$ and postcondition $\mathcal{R}$, we can prove this triple by proving the two programs individually, where the postcondition of C_1 and the precondition of C_2 overlap.

$$\frac{\vdash \{\mathcal{P}\}C_1\{\mathcal{Q}\} \quad \vdash \{\mathcal{Q}\}C_2\{\mathcal{R}\}}{\vdash \{\mathcal{P}\}C_1; C_2\{\mathcal{R}\}}\ \textsc{ht-seq}$$

In example 1 the HT-ASSIGN and HT-SEQ inference rules are used to verify the swap example. Here the precondition and postcondition are both defined as $\neg(x = y)$, corresponding to the assumption in line 1 and assertion in line 5 respectively. So, with C_{ex} our swap example, we proof that $\vdash \{\neg(x = y)\}C_{ex}\{\neg(x = y)\}$.

Example 1: *Program proof using Hoare logic.*

$$\dfrac{\dfrac{}{\vdash \{\neg(x = y)\}\ tmp := x;\ \{\neg(tmp = y)\}}\ \textsc{ht-assign} \quad \dfrac{\dfrac{}{\vdash \{\neg(y = tmp)\}\ x := y;\ \{\neg(x = tmp)\}}\ \textsc{ht-assign} \quad \dfrac{}{\vdash \{\neg(x = tmp)\}\ y = tmp;\ \{\neg(x = y)\}}\ \textsc{ht-assign}}{\vdash \{\neg(y = tmp)\}\ x := y;\ y := tmp;\ \{\neg(x = y)\}}\ \textsc{ht-seq}}{\vdash \{\neg(x = y)\}\ tmp := x;\ x := y;\ y := tmp;\ \{\neg(x = y)\}}\ \textsc{ht-seq}$$

Example 2: *Program proof using weakest precondition reasoning.*

$$
\begin{gathered}
\mathsf{wp}(tmp := x;\ x := y;\ y := tmp,\ \neg(x = y)) =^{\text{WP-SEQ}} \\
\mathsf{wp}(tmp := x,\ \mathsf{wp}(x := y;\ y := tmp,\ \neg(x = y))) =^{\text{WP-SEQ}} \\
\mathsf{wp}(tmp := x,\ \mathsf{wp}(x := y,\ \mathsf{wp}(y := tmp,\ \neg(x = y)))) =^{\text{WP-ASSIGN}} \\
\mathsf{wp}(tmp := x,\ \mathsf{wp}(x := y,\ \neg(x = tmp))) =^{\text{WP-ASSIGN}} \\
\mathsf{wp}(tmp := x,\ \neg(y = tmp)) =^{\text{WP-ASSIGN}} \\
\neg(y = x)
\end{gathered}
$$

Weakest precondition

Essential for automated verification of programs using Hoare rules is that the problem of determining $\vdash \{\mathcal{P}\}C\{\mathcal{Q}\}$ can be automated. Dijkstra has shown (initially for simple programs) that this can be done using *weakest preconditions* [Dij75]. A weakest precondition is, as the name suggests, the weakest set of logical assertions that need to be assumed before execution to ensure that a program satisfies a given postcondition after execution. Given a program, Dijkstra showed that we can use a function $\mathsf{wp}(C, \mathcal{Q})$ that defines this weakest precondition using structural recursion.

Concretely, using wp-reasoning for program verification can be done through the following property: $\vdash \{\mathcal{P}\}C\{\mathcal{Q}\} \iff \vdash \mathcal{P} \Rightarrow \mathsf{wp}(C, \mathcal{Q})$. The advantage of using wp-reasoning over Hoare logic to automatically verify programs is that we never need to find intermediate assertions - unlike Hoare logic, where this can be necessary.

If we want to prove our example using wp-reasoning, we need the wp-rules for **assignments** and **sequential composition**. Besides these two, basic wp-reasoning rules for **skip**, **conditionals**, **loops** and **consequence** also exist, but they are not listed here. For assignment the rule is very similar to the Hoare triple, where the wp for an assignment $x := e$ is the same as postcondition with all free occurences of x replaced by e:

$$\mathsf{wp}(x := e, Q) =^{\text{WP-ASSIGN}} Q[x/e]$$

Sequential composition is handled by first calculating the wp of the second program, and using this to calculate the wp of the first program:

$$\mathsf{wp}(C_1; C_2, Q) =^{\text{WP-SEQ}} \mathsf{wp}(C_1, \mathsf{wp}(C_2, Q))$$

The proof for the example swap program is shown in example 2. Here we show that the weakest precondition of the program is $\neg(y = x)$, and our initial assumption implies our weakest precondition, i.e. $\vdash \neg(x = y) \Rightarrow \mathsf{wp}(C_{ex}, \neg(x = y))$ with C_{ex} our example program. Recall that $\vdash \{\mathcal{P}\}C\{\mathcal{Q}\} \iff \vdash \mathcal{P} \Rightarrow \mathsf{wp}(C, \mathcal{Q})$, so we have proven that $\vdash \{\neg(x = y)\}C_{ex}\{\neg(x = y)\}$.

Loops and loop invariants

Loops often are an important part of programs, and both Hoare logic and wp-reasoning can be used to verify programs containing loops. Both the Hoare inference rule and the wp-reasoning rule for a loop **while** b **do** C require a *loop invariant* $\mathcal{P}$, which is typically specified by the programmer. For $\mathcal{P}$ to be a valid loop invariant it should hold before the first iteration of the loop, $\mathcal{P}$ is preserved by each iteration of the loop, and $\mathcal{P}$ holds after the loop has finished.

If $\mathcal{P}$ conforms to these three conditions, it can be used as a specification for the while loop in the Hoare rule. Though the weakest precondition of a loop program could in theory be calculated, due to the nature of this calculation, in practice this is often not feasible. This is why wp-reasoning often employs loop invariants as an alternative to this calculation.

To illustrate the concepts of loop invariants, two examples are given below. Here, the left example is correct while the loop invariant in the right example is not maintained. In the left example, the loop invariant states that the condition $i \geq 0$ must be preserved throughout the loop, and in the right example the invariant is $i < 10$. To check if the loop invariant holds, we need to check that it holds before the first iteration, and that the loop maintains it. Below each example we show why the loop invariants hold, or why not.

```
$i := 0$

loop invariant $i \geq 0$
while $i < 10$ do $i := i + 1$
```

The loop invariant is true before the iteration: i is initialised to 0, so $i \geq 0$. During the loop, the value of i only gets increased and so the value of i never will be below 0 (its starting value). This means that the loop invariant $i \geq 0$ is maintained by the loop.

```
$i := 0$

loop invariant $i < 10$
while $i < 10$ do $i := i + 1$
```

The loop invariant is true before the iteration: i is initialised to 0, so $i < 10$. However, the last iteration of the loops occurs when $i = 9$. The program enters the loop body, increments i to be 10, and exits the loop. The loop invariant $i < 10$ was *not* maintained by the loop, because after the last iteration $i = 10$.

3.1.2 Concurrent deductive verification

While Hoare logic is suitable for sequential programs, we might want to verify concurrent programs as well. The challenge of verifying concurrent programs lies in the fact that as soon as there are two or more program threads running in parallel we need to account for all possible interactions between all these threads.

Owicki-Gries rule

We can make things simpler by assuming that all threads are non-interfering. This means that the execution of assignments in one program will not change the state of any of the other threads. More specifically, if for some thread i we had $\vdash \{\mathcal{P}_i\}C_i\{\mathcal{Q}_i\}$, then this fact won't be changed by any assignment in another thread. In this case, and only in this case, we can use an addition to Hoare logic defined by Owicki and Gries [OG76], aptly named the *Owicky-Gries method.* This methods adds a rule for parallel composition, which we can add to the rule list from Hoare logic:

$$\frac{\vdash \{\mathcal{P}_1\}C_1\{\mathcal{Q}_1\} \quad \vdash \{\mathcal{P}_2\}C_2\{\mathcal{Q}_2\}}{\vdash \{\mathcal{P}_1 \wedge \mathcal{P}_2\}C_1||C_2\{\mathcal{Q}_1 \wedge \mathcal{Q}_2\}}$$

Concurrent Separation Logic

While the groundwork laid by Owicki and Gries can be useful in some contexts, more often than not we need to reason about threads independently of one another: both threads C_1 and C_2 in the rule need to know all interference information (which variables are assigned, for example) of the other thread and so they cannot exist independently of each other. For this reason, O'Hearn and Brookes extended the Hoare logic rules even further, with *Concurrent Separation Logic* (CSL) [O'H07, Bro07]. This extension can reason about concurrent programs without the need for non-interference. It does this by supporting the concepts of **ownership** and **disjointness**, and by defining rules for **advanced parallel composition**, **atomic programs** and several **heap manipulation operations**.

Furthermore, *Permission-Based Separation Logic* (PBSL) [AHHH15], an extension of CSL, adds a new syntax so that we can express the location of variables on the heap. If we want to express that the heap contains the value v on location l we can write $l \overset{\pi}{\hookrightarrow} v$. Here π is a rational number in the range $(0, 1]$, and it represents the fractional permission [Boy13]. The value of π defines the level of permission that is available, where $l \overset{1}{\hookrightarrow} v$ means write (and read) access, and any other valid value of π means that the program has only read access to that location on the heap. When considering several concurrent programs that all need some kind of access to a location on the heap, we need to ensure that the total sum of all permissions for that location does not exceed 1 at any point. We can show that if this is the case there is no data race (data race occurs when two or more threads access the same location on the heap simultaneously, and at least one of those threads has write permission).

Besides the notion of ownership, we can also express *disjointness of ownership* using the *separating conjunction.* The notation for an assertion containing a separating conjunction is $\mathcal{P} * \mathcal{Q}$, which is read as "$\mathcal{P}$ and separately $\mathcal{Q}$". It means that the two assertions $\mathcal{P}$ and $\mathcal{Q}$ do not both express write access to the same heap location. This notion can be used in PBSL to define a more advanced version of the rule for parallel composition, where the *resources* in the pre- and postconditions for the two programs are disjoint, thus ensuring non-interference. In PBSL, the resources of an assertion are defined by the combination of the heap and fractional permissions, and for them to be disjoint the sum of all permissions for each heap location cannot exceed 1.

3.2 VerCors

The VerCors tool set [BDHO17] is used in this thesis to apply deductive verification to an example graph algorithm: the set-based SCC model checking algorithm. VerCors can be used to reason about the behaviour of (concurrent) programs in OpenCL, OpenMP, Java, and a custom language PVL. This is done by annotating the programs with specifications such as preconditions, postconditions and loop invariants. Crucially, VerCors uses PBSL to reason about programs, which allows the user to verify concurrent software. In this section we briefly look at the architecture of the tool set, before exploring how deductive verification in VerCors (using PVL) is carried out.

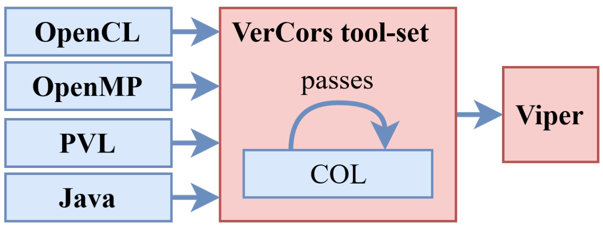


Figure 4: *Design of VerCors architecture [JOSH18].*

3.2.1 Architecture

In chapters 4 and 5 we discuss the application of VerCors to a graph algorithm as well as possible improvements to VerCors. For this reason we need to have an idea of how the VerCors architecture supports the concepts of deductive verification discussed in section 3.1. The architecture design of the tool set is shown in figure 4, and we first give an overview of this design, before explaining each part in more detail.

In the leftmost part of figure 4 the four input languages that are currently supported are listed: OpenCL, OpenMP, PVL and Java[1]. These input languages get translated to an *abstract syntax tree* (AST) in the intermediate abstract language *Common Object Language* (COL). In the middle part of the schema we carry out ”passes” that transform the COL AST. Finally, the program representation in COL AST is converted to a Silver AST (Silver is the language the tool's back end uses). This Silver AST is passed to the back end: *Verification Infrastructure for Permission-based Reasoning*[2] (Viper). This back end transforms the program into an *Satisfiability Modulo Theories* [BT18] (SMT) problem and tries to solve it.

Input languages

There are four input languages that are currently supported by VerCors. All languages are converted to the same intermediary language COL. This means that VerCors is easily extendable with new languages if desired: any input language could be used as long as there exists a translation from that language to a COL AST. After the conversion to COL all further work is universal and independent of the input language used.

Of the four languages, the *Prototype Verification Language* (PVL) is a custom language specifically designed for use by VerCors, and it is the language that is used in chapter 4. As the name suggests, it is designed to be used for easy prototyping of verification features. This is facilitated by the fact that PVL does not have a runtime environment, and so new verification features can easily be added. This has the additional benefit that almost all features of VerCors can be used in PVL, while this may not be the case for other languages. PVL is an object-oriented language, and natively supports program annotations for pre- and postconditions, and loop invariants. Syntactically, it resembles a language like Java, and the full syntax can be found on the official VerCors wiki page[3]. Because we use PVL in chapter 4, we explain the basic syntax of PVL programs and program annotations in section 3.2.2.

[1]At time of writing a subset of C is also supported, but not for all features.
[2]Main page for Viper: `https://www.pm.inf.ethz.ch/research/viper.html` (accessed 24-06-2021)
[3]PVL syntax wiki page: `https://vercors.ewi.utwente.nl/wiki#syntax` (accessed 24-06-2021)

Listing 1: *Example* `Clear` *class in PVL.*

```
class Clear {

    void clear(int[] A) {
        int i = 0;

        while (i < A.length) {
            A[i] = 0;
            i = i + 1;
        }
    }

}
```

COL and passes

All input languages get translated into COL, which is an abstract language, meaning that there is no syntax for it and it is only represented by an AST. After the initial translation from the input language to COL, several so-called *passes* are executed on the COL AST. Each pass is a transformation that changes, simplifies or in some way modifies the COL AST to facilitate the verification of the program.

Viper

Once all necessary passes on the COL AST have been executed, the AST is transformed to a Silver AST that can be used by the main back end of VerCors, called Viper. The Silver AST contains the program and all its specifications in the language Silver, which is the input language of Viper. Viper then processes the AST to turn in into an SMT problem which can be solved by the SMT-solver Z3. SMT-solvers try to find a model satisfying a set of first-order logic predicates. In our case, our predicates consist of our specifications (pre- and postconditions, loop invariants) and our program. The specific problem the SMT-solver tries to solve is to satisfy the *negation* of the predicates of the provided specification. This means that the solver tries to find a counterexample to our specification. So, if the solver *cannot* satisfy the negation of our predicates, there is no counterexample and the specification is met.

3.2.2 Deductive verification in VerCors using PVL

In listing 1 we show a basic example of a small PVL program[4]. This program clears an integer array, i.e. it sets all elements of the array to `0`. It consists of the class `Clear` (this encapsulation in a class is necessary, since PVL is an object-oriented language), and the method `clear` within that class. The method takes an `int` array `A`, and loops over the array using a while loop, setting each element of `A` to `0`. While this is a simple program, there are still properties that we might wish to verify. These properties may have to do with memory-safety or functionality. In the specification for this example, we want to make sure that:

[4]Based on the file `examples/demo2.pvl` from the official VerCors GitHub page: `https://github.com/utwente-fmt/vercors` (accessed 01-06-2021)

Listing 2: *Example `Clear` class in PVL with annotations.*

```
 1 class Clear {
 2
 3     context_everywhere A != null;
 4     context_everywhere (\forall* int j; 0 <= j && j < A.length;
 5         Perm(A[j], write));
 6     ensures (\forall int j; 0 <= j && j < A.length; A[j] == 0);
 7     void clear(int[] A) {
 8         int i = 0;
 9
10         loop_invariant 0 <= i && i <= A.length;
11         loop_invariant (\forall int j; 0 <= j && j < i; A[j] == 0);
12         while (i < A.length) {
13             A[i] = 0;
14             i = i + 1;
15         }
16     }
17
18 }
```

1. we avoid potential null pointer errors with respect to `A`,
2. we have the appropriate permissions for the data we need to access, and
3. the result of the `clear` method is a cleared array.

In order to verify that our program complies to this specification, we need to annotate the function with preconditions, postconditions and loop invariants, as shown in the next section.

Program Annotations

Even for a relatively small program the annotations can be quite numerous, as can be seen in listing 2, where annotations are added to the example program. For reference, the relevant keywords and concepts in the PVL annotation syntax are listed in table 1, along with brief explanations[5]. The rest of this section goes through the example specification, relating the items therein to the annotations in the program.

Item 1 in our specification is to avoid potential null pointer errors. In the `clear` method there is only one candidate variable that could be `null`, namely `A`. To avoid these errors, the annotation `context_everywhere A != null;` is added to the `clear` method (line 3). Now the verifier will check that `A != null` in the precondition and postcondition of the method, and also as a loop invariant for the while loop in line 12. This annotation gets successfully verified, because neither in the outer method body nor in the loop body is `A` ever set to `null`.

The next specification is that the program has the appropriate permission to the data it needs to access. In PVL this is done using the `Perm` clause. In line 4 and 5 the `\forall` quantifier is used to define write access *for every element in* `A` using `Perm(A[j], write)`. Note that a special variant of the quantifier is used: `\forall*`. If `A` had length `N`, the quantifier now expands to `Perm(A[1], write) * Perm(A[2], write) * ... * Perm(A[N-1], write)`,

[5]Full documentation of the PVL syntax and semantics can be found at: `https://vercors.ewi.utwente.nl/wiki#syntax` (accessed 01-06-2021)

where normally there would be an and operator (`&&`) in the place of the separating conjunction from CSL (`*`). Again, we use the `context_everywhere` keyword, since we want to retain the write access completely during the method.

The third and final specification demands that the result of the `clear` method is indeed a cleared array. In our case, a cleared array means that the value of every element in the array is `0`. We can encode this property as a postcondition for our method, as is done in line 6. Here too the `\forall` quantifier is used to make a statement about all elements in `A`, and the relevant boolean expression is simply `A[j] == 0`. However, the program will not verify by only annotating the method with a postcondition, since the verifier cannot ensure that the body of the method actually achieves what is necessary to satisfy the postcondition. In order to satisfy the postcondition, we need the loop invariant on line 11. This invariant states that before and after each iteration of the loop all elements in `A` *up to index* `i` (the iterator) have been cleared, i.e. are `0`. Intuitively this makes sense: initially, we cannot be sure that any of the elements are `0`, but this is not a problem since `i` gets initialised to `0`. After the first iteration, `i` is incremented but we also know the first element of `A` is `0`. We continue incrementing `i` and increasing our knowledge about which elements in `A` are cleared, until finally we reach the end of the array and we know all elements are now `0`. Using this knowledge we can exit the method and verify the postcondition.

There is one last loop invariant present in the example code that is not covered by the three specification items. This annotation in line 10 makes sure that the index `i` that is used in the while loop is never out of bounds of array `A`, ensuring the absence of memory safety issues. VerCors automatically checks for bounding problems that may cause these issues, and so the program will not verify without this loop invariant present. Note that the second part of the invariant states that `i <= A.length`, because the loop invariant should also hold *after the last iteration*. This is of course not enough to ensure that `i` is not out of bounds, namely if `i == A.length`. However, in combination with the condition of the loop in line 12, this is guaranteed for line 13. Additionally, this second part of the invariant allows us to conclude that `i == A.length` instead of only that `i >= A.length` when the loop terminates, which can help us with establishing stricter postconditions.

Table 1: *Basic keywords and constructs of program annotations in PVL.*

Keyword	*Explanation*
`assert`	The `assert` keyword followed by a boolean expression is interpreted as an assertion (the verifier will check if the assertion holds). It can be placed at any point in a method or function body, and the verifier checks if it holds *at that location.*
`assume`	This keyword is similar to the `assert`, only here the verifier *assumes* that the expression in the statement is true, so it does not check its validity.
`requires`	This keyword indicates a precondition statement. It can be placed before a method or function declaration. A `requires`-clause holds *before* the execution of the method or function body.
`ensures`	A postcondition statement is started with this keyword. Similarly to `requires`, it can be placed before a method or function declaration. An `ensures`-clause is assumed to hold *after* the execution of the method or function body.
`loop_ invariant`	Loop invariants can be specified using this keyword. The only place a loop invariant can be placed is before a `for` or `while` loop. The definition of a loop invariant is such that the statement in the invariant holds *before and after* each iteration.

Keyword	*Explanation*
`context`	To save space, we can use this keyword to indicate that the statement that follows is both a pre- and postcondition. The location of this keyword is the same as pre- and postconditions.
`context_everywhere`	This keyword is similar to `context`, but the statement is propagated to *all* loops in the function body as a loop invariant as well. The intention of this keyword (and often the its as well) is to indicate that the statement holds everywhere in the method body, though it can be broken and re-established again as long as it does not violate any conditions or invariants. Again, the location is before the function declaration.
`\forall`	The `\forall(varDecl; cond; expr)` construct is equivalent in meaning to the $\forall$ quantifier in mathematical notation. The syntax is somewhat similar to that of a `for` loop, where `varDecl` declares a variable, `cond` is a boolean expression describes the boundary conditions for the variable declared in `varDecl`, and `expr` is also a boolean expression, and it expresses the assertion in the statement. `\forall` quantifiers can be used in preconditions, postconditions and loop_invariants.
`\exists`	This construct is very similar to `\forall`, except its meaning is equivalent to the $\exists$ quantifier. The construct looks as follows: `\exists(varDecl; cond; expr)`.
`Perm`	This clause is equivalent to the ownership concept from PBSL, where `Perm(v, frac)`, with `v` some variable and `frac` the fractional permission, is equivalent to $l \overset{\pi}{\hookrightarrow} v$ (l is not relevant here, since it is managed automatically). All restrictions with respect to the fractional permission apply here as well: no two or more concurrent threads can have write permission at the same time.
`**`	The `**` operator is equivalent to the separating conjunction operator in PBSL. It can be used between `Perm` clauses, and the special quantifier `\forall*` is used to specify that for all applicable locations on the heap, they are disjoint.
`\old`	`\old(expr)` can be used in postconditions to indicate the value of `expr` *before* the execution of the method.
`\result`	Like `\old`, this keyword can be used in postconditions. It represents the return value of the method.
`ghost`	Every statement that is preceded by the `ghost` keyword is so-called *ghost code*. This code is used as helper code, and not compiled (if this is applicable). Ghost code generally does not interact with the normal code.
`given`	If the contract of a method contains a `given type var` keyword, followed by a parameter, this means that the method is extended with a *ghost parameter* `var` of type `type`. This parameter can be used in ghost code in the method as usual, and it can be assigned using the `with` keyword, and retrieved with the `then` keyword.
`yields`	The presence of the keyword `yields type var` in the method contract indicates that the method returns the ghost variable `var` of type `type`.
`pure`	This is a keyword that indicates that a method is a pure function. A pure function body consists of a single expression. Pure functions have the form `modifiers pure return_type name(args) = expr;`, where modifiers can be keywords like `public` or `static`. Pure functions must be without side effects, i.e. not alter the (ghost) program state.

Keyword	*Explanation*
`inline`	Finally, by adding the keyword `inline` to a method, a call to that method effectively gets replaced by the method body when the code is analysed. The keyword is placed as a modifier in a method declaration. There are some constraints on the method body, for example it should be sufficiently simple and contain no recursion.

4 VERIFICATION OF A SEQUENTIAL SET-BASED SCC ALGORITHM

The first research question outlined in the introduction of this thesis is RQ1: *What techniques are involved with proving the correctness of a high-level model checking algorithm using the VerCors tool set?*. To answer this question, we need to find a suitable algorithm, understand how it works and formalise the desired properties we want to verify.

Oortwijn has done work on a similar subject, namely verifying a parallel model checking algorithm using depth-first search (NDFSS) [Oor19]. Their goal was to explore how to use PBSL to automatically verify high-level graph algorithms, and they also used he VerCors tool set. They recommended several candidate algorithms for future work that continues this line of inquiry, including (parallel) model checking algorithms that are SCC-based.

Several factors determine which algorithm to use for this project. First and foremost, we need a manual proof for the soundness and completeness properties of the algorithm, or at least an outline thereof. Furthermore, the scope (or complexity) of the algorithm needs to be compatible with the scope of the project in terms of effort, time and expertise needed to complete the verification. Lastly, the verification effort should be able to adequately answer the first research question RQ1, and contribute significantly to the second question RQ2.

When taking into account these criteria, and looking at a list of suitable candidates (see appendix A) we choose a sequential set-based SCC algorithm for this project. This algorithm is outlined by Bloemen et al. in [BBDL$^+$18] and [Blo19]. Here, the definitions for the completeness and soundness properties as well as a brief proof outline are also given. This algorithm is not parallel, which means that verifying it does not necessarily shed light on the use of VerCors in a parallel setting, but it does employ enough graph-related concepts to be of use for answering the two research questions.

The remainder of this chapter contains the following: section 4.1 introduces the set-based SCC algorithm, section 4.2 briefly outlines the correctness criteria, section 4.3 sketches the proof of the completeness and soundness properties, and section 4.4 shows the verification effort and the encoding of the algorithm in VerCors.

4.1 Set-based SCC algorithm

As mentioned previously, the algorithm that is verified using VerCors is the set-based SCC algorithm as described by Bloemen [Blo19]. This algorithm computes all SCCs of a generalised Büchi automaton. It is based on the DFS search principle and runs in linear time. These kinds of algorithms are based on the work of Tarjan [Tar72], who proposed an algorithm (Tarjan's algorithm) that partitions the set of states into SCCs. The versions of the algorithm in [BBDL$^+$18] and [Blo19] use the principles of Tarjan's algorithm, along with a union-find database structure, as an emptiness check for TGBAs. This particular version of an SCC-based algorithm is based on Munro's algorithm [Mun71] while collapsing SCCs immediately like in [Dij76] and [RDLKP13]. Union-find structures partition the complete set of states into subsets, and each set has one representative (i.e. root) of the set. Using a union-find structure, we can create new partitions with some arbitrary state in that partition as a root, we can find the root state of any state, we can check if two states are in the same set and we can merge partitions. Several variants of this union-find structure have been proposed and properties have been explored by [Tar75, TvL84] and notably [Blo19] (for concurrent applications).

In the rest of this section, the algorithm is explained in more detail. This is eventually done line-by-line in section 4.1.2, but first some concepts that are specific to this algorithm are explored in section 4.1.1. At the end of the section, we show an example of the algorithm in action on a small graph.

4.1.1 Algorithm-specific concepts

There are some concepts used in this algorithm that are specific to it, and need some introduction. These concepts are the Explored and Visited sets, the stack of roots and the union-find structure. Below we explain these concepts and briefly describe their use in the algorithm.

Explored and Visited sets

The strictly increasing Explored and Visited sets keep track of the exploration-states of the states in a graph. If a state v is *completely explored*, i.e. we used DFS to explore all its successors, then $v \in \mathsf{Explored}$. However, if we have *visited* a state v, but possibly not yet explored all its successors, then $v \in \mathsf{Visited}$. As a consequence, $\mathsf{Visited} \subseteq \mathsf{Explored}$.

These two sets are used to decide what actions are taken in each iteration of the algorithm. Depending on the exploration-state of the successors of a state, we can either ignore the successor (if it is already in Explored), visit the successor (if not yet in Visited), or combine the partitions with those of other states in the same SCC (i.e. collapse the SCC, if not in Explored but already in Visited). Using these two sets, we can also define two more exploration-state concepts that are useful in the correctness argument in 4.3:

1. $\mathsf{Unseen} = V \setminus \mathsf{Visited}$ where V is the set of all states. This is the set of states that have not been encountered before in the DFS.
2. $\mathsf{Live} = \mathsf{Visited} \setminus \mathsf{Explored}$, the set of all *live* states that are visited by the DFS, but not yet part of a completely explored SCC.

Stack of roots

The algorithm keeps a stack R. This stack is a subset of the program stack (the DFS search stack), and all states on R are in the same order as on the program stack. R does not necessarily include all states in the program stack, but it nonetheless maintains the property that each state on R has a path to all states higher on the stack. We define three operation on the stack: $\mathsf{Push}(v)$, $\mathsf{Pop}()$ and $\mathsf{Top}()$. The $\mathsf{Push}(v)$ operation pushes the state v onto the stack, the $\mathsf{Pop}()$ operation returns the top state and removes it from the stack, and the $\mathsf{Top}()$ returns the top state while *not* removing it from the stack. This stack is used to collapse an SCC when a cycle is detected.

Union-find structure

For these algorithms a so-called union-find structure is used. This data structure divides a set S into disjoint partitions, where each partition has one representative (this representative is also in S). This data structure can be implemented as a simple sequence of integers, where the index is the identifier and the value points to the index containing the representative, or a map where the keys are the representatives pointing to a set of states as the value. Alternatively, it can be implemented as an equivalence relation or a cyclic queue, like in [Blo19]. This last implementation has several advantages when it comes to concurrent access to the data in the union-find.

There are several operations possible on a union-find structure: new partitions can be created with some state as a root, the root of any state can be found, we can check if two states are in the same set, and partitions can be merged. If implemented in a specific way, all these operations can be done in quasi-constant time [BBDL+18] as well as concurrently, making the union-find a suitable data structure for parallel algorithms.

In the set-based SCC and UFSCC algorithms (the latter is the parallel version of set-based SCC), the union-find is used to partition the states in the Büchi automaton in such a way that eventually each partition is an SCC.

4.1.2 Pseudocode

Algorithm 1 shows the pseudocode of the sequenatial set-based SCC algorithm as shown in [Blo19]. The actual implementation of the algorithm in PVL is somewhat different and will be discussed in section 4.4. Below, in table 2, the algorithm is explained line-by-line.

Table 2: *Line-by-line explanation of algorithm 1.*

Line	*Explanation*
1	Initialise the union-find structure for SCCs. Every state v maps to a set $\{v\}$ containing only itself using the function $\mathcal{S} : V \to 2^V$ to represent the union-find structure.
2	Initialise the set of completely explored states, called Explored, to the empty set.
3	Initialise the set of states that have been visited (but not necessarily completely explored), called Visited, to the emtpy set. *Note that Explored and Visited are both strictly increasing.*
4	Initialise the stack of SCC 'roots', called R, to an empty sequence. This stack contains a subset of states from the program stack (the stack of states already encountered, but not yet backtracked). The states on R have the same order as the states on the program stack, but they have the additional property that all states on the stack have a path to all other states that are higher on the stack. There are three operations defined on R: Push, Pop and Top.

Line	*Explanation*
5	This is the main function call, with the initial state v_0 as argument.
6	The start of the function SetBased.
7	Add v (the current state) to the Visited set, indicating that SetBased is called with v but has not yet finished.
8	Push v onto R.
9	Iterate over all successors w of v. For every successor do lines 10-16:
10	Check if the current successor is already in the Explored set. States get added to this set in line 19, so this can only happen if during the exploration of another successor w' of v using DFS w was encountered, *and* the SCC to which w, w' and possibly (but not necessarily) also v belong is completely explored already.
11	If the successor is not yet explored, the algorithm needs to check if it is in the Visited set. If not, we recursively call SetBased on the successor.
12	If this line is reached, it means that the successor is in Visited but not in Explored. Similarly to the normal DFS algorithm, this means that a cycle is found. In this case, the algorithm collapses the states, which is done in lines 13-15. We do this on the premise that there exists *some* state w', such that w' is on the stack R, and the current successor w is in $\mathcal{S}(w)$
13	We keep collapsing states until the partitions to which v ($\mathcal{S}(v)$) and w ($\mathcal{S}(w)$) belong are equal to each other; they belong to the same partition.
14	We pop the top element from R, and assign it to r.
15	$\mathcal{S}(r)$ and $\mathcal{S}(R.\mathsf{Top}())$ are collapsed. These last two lines are repeated until the condition on line 13 is not met.
16	After the states are collapsed the algorithm reports $\mathcal{S}(v)$ as an FSCC, since it is a cycle.
17	If, after all children are visited, v is on top of the stack this means that the SCC to which v belongs has been completed. This can intuitively be explained by the fact that if v is on top of the stack, all cycles containing v have been collapsed in the union-find.
18	The SCC containing v, $\mathcal{S}(v)$ can be reported.
19	The states of the SCC are added to Explored.
20	v is removed from the stack R.

4.1.3 Example

In figure 5 we show an example run of the set-based SCC algorithm on a simple graph consisting of three states, a, b and c. The figure is read top to bottom, left to right. In the figure we keep track of the values of the variables v, w and r. These variables can have any of the three states as a value. Aditionally, the stack R and union-find S are shown, as well as the exploration-state (in Visited or Explored) of a, b and c: the state is white if in Unseen, blue if in Visited, and red if in Explored. The search order is indicated with dotted arrows, and finally we indicate which line in the code in algorithm 1 affects the change displayed.

Initially the stack R is empty, and every state has its own partition in the union-find S, as shown in step **1**. We then visit the initial state $v = \mathsf{a}$ in step **2** and push it on R. We choose our first successor $w = \mathsf{b}$. Because b is not yet visited we reach line 11, and the DFS continues to step **3** by recursively calling the algorithm.

In this new call $v = \mathsf{b}$, and we again push it to the stack. Our next successor will be $w = \mathsf{c}$, and we reach line 11 again and continue the DFS search in step **4**. Similar to the first few steps, we

Algorithm 1: Set-based SCC Algorithm

input : Graph $G = (V, E)$ and initial state $v_0 \in V$.
output: Disjoint set union $\mathcal{S}$ of strongly connected components in G.

```
1  $\forall v \in V : \mathcal{S}(v) := \{v\}$
2  Explored := $\emptyset$
3  Visited := $\emptyset$
4  $R := \emptyset$
5  SetBased($v_0$)
6  function SetBased($v$)
7      Visited := Visited $\cup \{v\}$
8      $R$.Push($v$)
9      forall $w \in$ Succ($v$) do
10         if $w \in$ Explored then continue
11         else if $w \notin$ Visited then SetBased($w$)
12         else
13             while $\mathcal{S}(v) \neq \mathcal{S}(w)$ do
14                 $r := R$.Pop()
15                 Unite($r, R$.Top())
16             report FSCC $\mathcal{S}(v)$
17     if $v = R$.Top() then
18         report SCC $\mathcal{S}(v)$
19         Explored := Explored $\cup\, \mathcal{S}(v)$
20         $R$.Pop()
```

push $v =$ c onto R, and choose $w =$ b our next successor in the DFS search. However, since b is already in the Visited set (coloured blue), we do not reach line 11 but instead continue to the else-clause in line 12.

In step **6** we can see the body of this clause in action: we continue to merge the partitions in S that the two top states on R belong to (and then popping the top state of R) until they are the same. In this case, we need only one iteration of the while loop on lines 13-15 to achieve this, and afterwards we have popped $r = c$ off R and merged the partitions of c and b. We then exit the loop started in line 9 and skip lines 17-20, since the condition is not met.

This means the end of the call is reached, and we backtrack the DFS to the previous call where $v =$ b, which is step **7**. Here we do enter the if-statement started in line 17, and add all states in the partition of b in S to the Explored set. We then backtrack the search again so that $v =$ a.

From this point we still have another successor of a to explore, namely $w =$ c, shown in step **8**. However, we immediately reach the continue statement in line 10, meaning that we end the loop of line 9 and continue to step **9** where a is popped off R and added to Explored.

After the algorithm has finished all states are Explored, and S is partitioned in such a way that each partition is a maximal SCC.

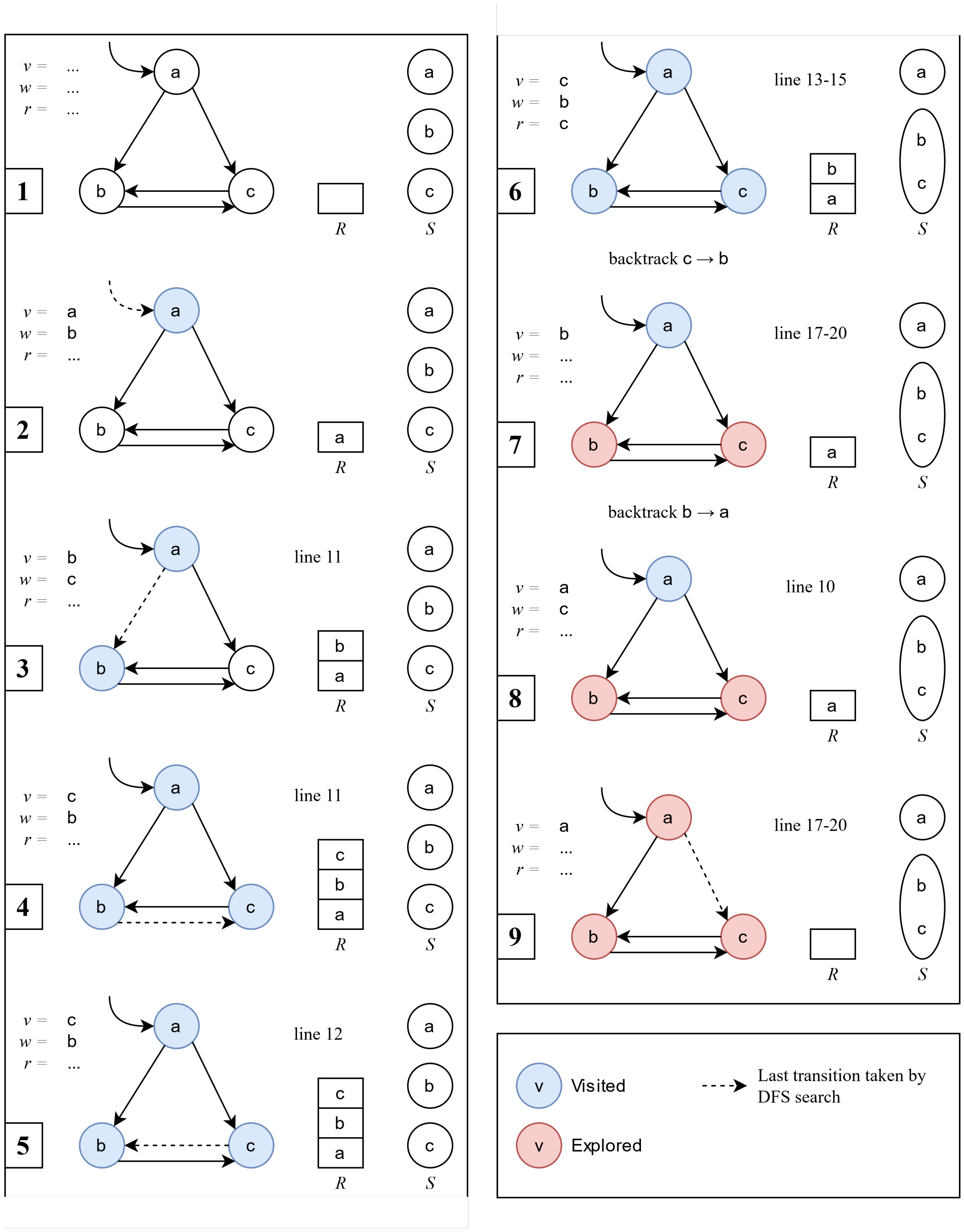


Figure 5: *Example of the set-based SCC algorithm on a small graph.*

4.2 Correctness criteria

We want to verify that the model checking algorithm works *correctly*, which implies that we want to prove the *soundness* and *completeness* (and possibly termination) of the algorithms. Bloemen [Blo19] defines the soundness and completeness of this algorithm as follows: soundness implies that all reported components are strongly connected, and completeness implies that all reported components are *maximal* (see 2.1.5) and that all reachable SCCs have been found.

Concretely, this means that we need to prove the following properties on the resulting union-find (disjoint set union) structure S:

1. For all partition sets $c \in S$, c is strongly connected.
2. For all partition sets $c \in S$, c is maximal.
3. All reachable SCCs are in S.
4. (Optional: the algorithm terminates.)

The algorithm should be annotated with conditions and invariants that prove these properties. Additionally, the definitions for valid union-find structures, stacks, different types of SCCs (PSCC, FSCC, SCC) and paths need to be defined. Auxiliary lemmas may also be necessary to prove some of the properties about the aforementioned concepts.

4.3 Proof outline

In [Blo19], Bloemen gives a proof outline for the correctness proof of the algorithm. This outline is useful to develop the verification of the algorithm using VerCors, since the ideas can be transferred with relative ease. Most notably, there are two important properties that translate very well to the deductive verification method and help the effort greatly. The rest of this section presents Bloemen's proof outline.

The first step is to provide a partition of the states in the graph into three (separate) sets, where $\forall v \in V$, with V the set of (reachable) states in a graph:

1. $v \in \mathsf{Explored}$. This means that v is completely explored by the DFS, and it is part of a complete SCC. Aditionally, $\mathsf{Explored} \subseteq \mathsf{Visited}$.
2. $v \in \mathsf{Unseen}$, with $\mathsf{Unseen} = V \setminus \mathsf{Visited}$. If v is in this set, it means that it has not been encountered before by the DFS.
3. $v \in \mathsf{Live}$, with $\mathsf{Live} = \mathsf{Visited} \setminus \mathsf{Explored}$. This set contains all states that are already visited by the DFS, but not yet completely explored: the sets are "live".

Initially each state is Unseen. When the DFS encounters the state, it becomes Live, and eventually the state will be $\mathsf{Explored}$. After the algorithm is finished all states are $\mathsf{Explored}$.

The first important property of the algorithm is that it ensures that all supernodes (or partitions) that the states on the stack R belong to are disjoint and together contain all Live states:

$$\biguplus_{v \in R} S(v) = \mathsf{Live} \tag{4.1}$$

Here, $\uplus$ represent the disjoint union of sets. Additionally, all Live states have a unique representation on R (a result of the disjointness of the union in equation 4.1):

$$\{S(v) \cap R \mid v \in \mathsf{Live}\} = \{\{r\} \mid r \in R\} \tag{4.2}$$

In other words, the supernode $S(v)$ of each Live state v has exactly one representative r in the union $S(v) \cup R$, where $r \in R$ and $S(v) = S(r)$. These two equations aid the correctness argument explained below.

From the above it follows that if w is a successor of v, and w is Live, then $\exists w\prime : S(w\prime) = S(w)$. In other words, there is a representative $w\prime$ on the stack for every Live state. In lines 12-16 the top two states on R are united until $w\prime$ is encountered. In that case, $S(w\prime) = S(v) = S(w)$, and since by definition $w\prime$ has a path to v (because of the nature of DFS), all united components are strongly connected (guaranteeing the soundness of the algorithm). Because of equation 4.2, we know that there is no other Live state that is part of the united component. This fact guarantees completeness, i.e. that all reported SCCs are maximal.

Eventually all states are marked $\mathsf{Explored}$ in line 19, when v is the top state on R. If this is the case, it means that there are not states lower on R (so, predecessors) that v could be united with, and the SCC is complete and can be reported.

4.4 Verification of the algorithm

In this section the verification effort will be explained. This is done by looking at the code of the PVL file used [1], and detailing the functionality and purpose of each part of the code. Aditionally, we explain the way each concept related to the algorithm is encoded in PVL (for example the various data-structures and graph properties). At several points, some code is omitted and replaced by `[...]` for brevity. Lines are only omitted if it is not immediately necessary to understand the code and the concepts behind it. The line numbers in the code snippets relate to the file directly, so the omitted code can easily be looked up in the complete file, if desired.

The file does not completely verify the correctness of the set-based SCC algorithm. It follows the proof outline in section 4.3 up to equation 4.2. The rest of the verification has yet to be carried out. This thesis does not include this remaining part because of time constraints, but section 4.4.11 briefly describes the work that still has to be done to complete the verification, and suggestions on how to go about this.

The complete file can be run with VerCors version 1.3.0 (the latest version at time of writing). However, two small changes need to be made to the source code: support for the syntax of tuples and set comprehension, and a pass on the COL AST needs to be disabled. This pass does not influence the correctness of the verification procedure, so it can be safely skipped.

[1]The full code can be found at: `https://github.com/HanHollander/SetBasedSCC`

Listing 3: *Global state.*

```
int N;                         // Size of graph
seq<seq<boolean>> G;           // Adjacency matrix of graph
seq<int> S;                    // Union-find structure
seq<int> R;                    // Stack of roots
set<int> Explored;             // Set of explored states
set<int> Visited;              // Set of visited states
```

Listing 4: *Stack.*

```
// determines whether R is a valid stack, represented by a sequence
static inline pure boolean Stack (int N, seq<int> R) =
    (\forall int i; 0 <= i && i < |R|; (\forall int j; i < j && j < |R|;
       R[i] != R[j])) &&
    (\forall int i; i in R; ru(i, N));

// pushes state v to the stack R
[...]
static pure seq<int> push(int N, seq<int> R, int v) =
    R ++ v;

// returns and removes last state (highest index) of/from R
[...]
static pure tuple<seq<int>, int> pop(int N, seq<int> R) =
    tuple<seq<int>, int> {R[0..|R| - 1], R[|R| - 1]};

// returns last state (highest index) of R (peeks at top of R)
[...]
static pure int top(int N, seq<int> R) =
   R[|R| - 1];
```

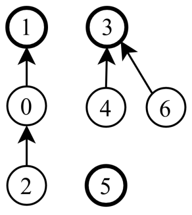
4.4.1 Global state and helper functions

The global state kept by the PVL program (listing 3) consists of six variables declared at class level. It almost directly corresponds to the global state in algorithm 1, with two additions, namely the graph `G` and its size `N`. The graph is encoded as an adjacency matrix, where `G[i][j]` is true iff there is an transition from state `i` to `j`. Properties about the global state are defined later in the program.

States are identified by an unique integer, so the stack `R`, the union-find `S` and the two exploration state sets `Visited` and `Explored` can be declared as sequences or sets of integers. Properties of these data structures are be defined later in the program.

Several helper functions are defined as well. These functions are *inline* (see section 3.2.2), and are used to cut down on the length of specification in the rest of the program. They define disjointness of sets and several range functions for variables. For example, the *range up to* function `ru(x, N)` gets expanded to `0 <= x && x < N` when the verifier is run.

index	0	**1**	2	**3**	4	**5**	6
value	1	**1**	0	**3**	3	**5**	3



(a) *Sequence* **(b)** *Tree*

Figure 6: *An example of a union-find represented as a sequence and a tree. Each sub-tree is a partition in the union-find. The representatives of each partition are marked bold.*

4.4.2 Stack

The stack of states (listing 4) is represented as a sequence of integers (`seq<int> R`), where the values in the sequence correspond to the state identifiers. The higher the index of an integer in the sequence, the higher the state is on the stack. There are some constraints on a valid stack for a graph of size `N`, in the `Stack` predicate in line 34: all elements on the stack are unique (line 35), and all elements have a valid value in the range $[0, \texttt{N})$ (line 36, by `ru(i, N)`).

Three operations are defined on the stack `R`: `push` (line 46), `pop` (line 57) and `top` (line 65). The implementation of `push` is straightforward: it appends the state `v` to `R`. This operation requires the stack to be valid both before and after the operation, so `v` should not yet be in `R` when it is pushed. The `pop(N, R)` operation returns a tuple containing the popped state and the new stack. Here too, the stack should be valid in both the pre- and postcondition; the latter is ensured by checking that the size of the stack is at least `1` in the precondition. Similarly, this precondition is also required for the `top(N, R)` operation. The difference with `pop` is that we do not return a new stack, and the input stack is not modified.

4.4.3 Union-find

Like the stack, the union-find data structure (listing 5) is represented by a sequence of integers (`seq<int> S`). Here the sequence represents a tree structure where for union-find `S`, `S[i]` is the successor of `i`. Each sub-tree represents a partition in the union-find, and each root node is such that `S[i] == i`. To illustrate this, both the sequence and tree representation of a simple union-find are shown in figure 6. This union-find has 3 partitions: $\{0, \mathbf{1}, 2\}$, $\{\mathbf{3}, 4, 6\}$ and $\{\mathbf{5}\}$. The states marked bold are the *representatives* (roots) of each partition, and each partition can be uniquely identified by this representative.

The constraints on the union-find `S` are given by the `UnionFind(N, S)` predicate in line 73. It defines that the size of `S` is exactly `N` (the size of the graph), and all elements have a valid value in the range $[0, \texttt{N})$. Another constraint that we could implement is that the sequence has to be a valid tree, i.e. there are no cycles in the resulting graph. However, this complicates the predicate (and verification as a whole) and up to this point in the verification it has not been needed to explicitly verify this property. Intuitively however, if the union-find initially does not contain cycles this property of the union-find is easily shown to be maintained throughout the algorithm. This is because values are only ever changed to root values, and so a cycle can never arise. The way this is guaranteed is two-fold, and has to do with the implementation of the `unite` operation and the way this operation is called in the actual algorithm.

Listing 5: *Union-find.*

```
 72 // determines whether S is a valid union-find
 73 static inline pure boolean UnionFind(int N, seq<int> S) =
 74     |S| == N &&  // size constraint on S
 75     (\forall int i; ru(i, N); ru(S[i], N));  // bounds on elements of S
 76
 77 // key invariant for the union-find data structure (unused)
 78 requires UnionFind(N, S);
 79 static inline pure boolean UFInvariant(int N, seq<int> S) =
 80     (\forall int v; 0 <= v && v < N; (\forall int w; 0 <= w && w < N;
 81         (w in part(N, S, v)) == (part(N, S, v) == part(N, S, w))));
 82
 83 // finds the representative of v in S
  ⋮ [...]
 88 static pure int rep(int N, seq<int> S, int v) =
 89     S[v] == v ? v : rep(N, S, S[v]);
 90
 91 // unites the partitions containing v and w
  ⋮ [...]
 98 ensures (\forall int i; ru(i, N) && rep(N, S, w) == {:rep(N, S, i):};
        rep(N, \result, i) == rep(N, S, v));
 99 ensures (\forall int i; ru(i, N) && rep(N, S, w) != {:rep(N, S, i):};
        rep(N, \result, i) == rep(N, S, i));
100 static seq<int> unite(int N, seq<int> S, int v, int w) {
101     seq<int> T = S[SB.rep(N, S, w) -> SB.rep(N, S, v)];
102     if (rep(N, S, w) == rep(N, S, v)) {
103         assert T == S;  // Trivial case
104     } else {
105         lemma_7_uf(N, S, T, v, w);
106         lemma_8_uf(N, S, T, v, w);
107     }
108     return T;
109 }
110
111 // finds the partition in S that v is in
  ⋮ [...]
116 static pure set<int> part(int N, seq<int> S, int v) =
117     setCompPart(N, S, v);
  ⋮ [...]
127 static pure set<int> setCompPart(int N, seq<int> S, int v);
```

The program has three operations that can be used on a union-find: `rep`, `unite` and `part`. The `rep(N, S, v)` operation in line 88 returns the representative of `v` in `S`. It requires that `S` and `v` are valid with respect to `N`.

The `unite(N, S, v, w)` operation in line 100 unites the partitions of `v` and `w`. It too requires that `S` and `v` are valid, and it ensures that the result is again a valid union-find, along with two properties that are proven by lemma 4 and 5, in lines 98 and 99. The full explanation of these lemmas is given in section 4.4.9 and its use is explained in section 4.4.10, but the intuitive meaning is as follows: after the partitions of `v` and `w` are united, all states that previously had the same representative as `w`, now have the same representative as `v` (lemma 4), and the rest of the union-find is unchanged (lemma 5).

An example of the `unite` operation on the example union-find of figure 6 can be seen in figure 7. One of the ways this union could happen is by executing `unite(1, 3)`, so by uniting the roots. However, if for example `unite(2, 6)` was executed, the result would have been the same. Our implementation, as can be seen in line 101, changes the value with the representative of the second state argument as its index to the representative of the first state argument. In an efficient implementation we would also want to *flatten* the tree (make all non-root nodes point directly to the root) to be more efficient in the long run. For example, this flattening helps with the recursive representative operation `rep`, since we only ever need one iteration.

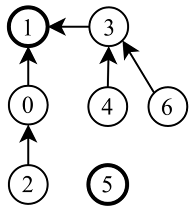


Figure 7: *Example of the unite operation. Representatives are marked bold.*

As mentioned previously, the implementation of `unite` implicitly preserves the non-cyclic nature of the union-find. This can be seen in line 101: the value at index `rep(N, S, w)` is changed to `rep(N, S, v)`, and the value at index `rep(N, S, v)` is by definition also `rep(N, S, v)`. If there were no cycles present previously, none have been introduced: the only value changed is the value at index `rep(N, S, w)`, and it is changed to point to `rep(N, S, v)`, which points to itself.

Lastly, the `part(N, S, v)` operation in line 116 returns the partition that `v` is in. Again, `S` and `v` need to be valid with respect to `N`. The operation ensures that the result is indeed the partition of `v` by having two postconditions. The first postcondition expresses that all elements with the same representative as `v` are in the returned partition, and the second postcondition verifies that all states in the returned partition have the same representative as `v`. The implementation of this operation is not complete: it is simply a call to the workaround function `setCompPart(N, S, v)`. PVL supports set comprehension expressions, and this would be very well suited here. However, at the time of writing it is not possible to carry over the properties of arguments, for example `S`, that were established in the precondition to the body of the set comprehension expression. If another method or function, with certain preconditions, is then used in the set comprehension body this can cause problems. The current workaround should not have any negative effects on the verification of the algorithm, since essentially `setCompPart` can be seen as a definition of a set comprehension. More specifically, it defines a set comprehension over all states, with the condition that the states need to have the same representative as some state `v`.

As a side-note, in line 79 another invariant over the union-find is defined. This invariant is currently unused, but still potentially relevant in the part of the proof that is yet to be done. From [Blo19], it expresses the following invariant over valid union-find structures for a graph with size `N`: $\forall v, w \in [0, \texttt{N}) : w \in \texttt{part(N, S, v)} \iff \texttt{part(N, S, v)} = \texttt{part(N, S, w)}$.

4.4.4 Graph-related concepts

There are some global graph-related concepts (listing 6) that we want to express. Two concepts define constraints on components of a graph, namely `Component(N, C)` (line 134), which constraints the elements of a set `C` to be in bounds with respect to `N`, and `NonEmptyComponent(N, C)` (line 138), which has the additional constraint that it can not be empty. There is no need to explicitly declare each element in a component unique, since this is an inherent property of sets in PVL. The third global concept, `AdjacencyMatrix(N, G)` in line 142, defines `G` a valid adjacency matrix of size `N`, where it should have the right size ($\texttt{N} \times \texttt{N}$).

Listing 6: *Graph-related concepts.*

```
133 // determines whether C is a component in a graph with size N
134 static inline pure boolean Component(int N, set<int> C) =
135     (\forall int x; x in C; 0 <= x && x < N);
136
137 // determines whether C is a component of at least size 1
138 static inline pure boolean NonEmptyComponent(int N, set<int> C) =
139     Component(N, C) && |C| > 0;
140
141 // determines whether G is an adjacency matrix of size N
142 static inline pure boolean AdjacencyMatrix(int N, seq<seq<boolean>> G) =
143     |G| == N && (\forall seq<boolean> e; e in G; |e| == N);
```

Listing 7: *Algorithm-related concepts.*

```
149 // defines the set of all states V(N)
150 ensures (\forall int j; j in \result; range(j, 0, N));
151 ensures (\forall int j; range(j, 0, N); j in \result);
152 static pure set<int> V(int N) =
153     set<int> {i | int i, int N <- [N]; SB.range(i, 0, N)};
154
155 // defines the Unseen() set
156 requires Perm(N, 1\2) ** Perm(Visited, 1\2);
157 pure inline set<int> Unseen
158     V(N) - Visited;
159
160 // defines the Visited() set
161 requires Perm(Visited, 1\2) ** Perm(Explored, 1\2);
162 pure inline set<int> Live() =
163     Visited - Explored;
```

4.4.5 Algorithm-related concepts

Some useful algorithm-related concepts (listing 7) that are used in the proof of the algorithm, but are not necessarily part of it are the set of all states `V(N)`, and the derived sets `Unseen` and `Live`. The definition of `V(N)` in line 152 is simply the set of all integers in the range $[0, \texttt{N})$. The other two sets are derived from `Explored` and `Visited`, which were previously defined in 4.4.1. The first, `Unseen` in line 157, is comprised of all states that are not yet visited, so `V(N)` $\setminus$ `Visited`. The second derived set is `Live` in line 162, and this is the set of all states that are currently being explored. In other words, `Live` is the set of all states that are already visited, but not yet fully explored: `Visited` $\setminus$ `Explored`.

4.4.6 Paths

Paths are represented as sequences (listing 8), similar to definition 1 in section 2.1.5. There are two types of paths defined in the program, normal paths and fitting paths. Along with the definition functions, we also declare existential quantifier functions. Each function has the same precondition, namely that the graph `G` should be valid with respect to `N` (using `AdjacencyMatrix`).

Listing 8: *Paths.*

```
169 // determines whether P is a path from x to y in the graph G.
170 requires AdjacencyMatrix(N, G);
171 static pure boolean Path(int N, seq<seq<boolean>> G, int x, int y,
        seq<int> P) =
172      0 <= x && x < N && 0 <= y && y < N &&
173      0 < |P| && P[0] == x && P[|P| - 1] == y &&
174      (\forall int j; 0 <= j && j < |P|; 0 <= P[j] && P[j] < N) &&
175      (\forall int j; 0 <= j && j < |P| - 1; G[P[j]][P[j + 1]]);
176
177 // existential quantification over paths in G of length at least len.
178 requires AdjacencyMatrix(N, G);
179 static pure boolean ExPath(int N, seq<seq<boolean>> G, int x, int y, int
        len) =
180      (\exists seq<int> P; len <= |P|; Path(N, G, x, y, P));
181
182 // determines whether P is a path from x to y in the graph G visiting only
        states in C.
183 requires AdjacencyMatrix(N, G);
184 static pure boolean FittingPath(int N, seq<seq<boolean>> G, int x, int y,
        seq<int> P, set<int> C) =
185      Path(N, G, x, y, P) &&
186      (\forall int v; v in P; v in C);
187
188 // existential quantification over fitting paths in G of length at least
        len.
189 requires AdjacencyMatrix(N, G);
190 static pure boolean ExFittingPath(int N, seq<seq<boolean>> G, int x, int
        y, int len, set<int> C) =
191      (\exists seq<int> P; len <= |P|; FittingPath(N, G, x, y, P, C));
```

A normal path is defined in line 171 by `Path(N, G, x, y, P)` as follows: `P` is a path in `G` iff `x` and `y` are valid with respect to `N`, the length of the path is at least 1, the first element in the path is `x`, the last element is `y`, all elements in the path are valid with respect to `N`, and for all successive states `P[j]` and `P[j + 1]` in the path, the value in the adjacency matrix `G[P[j]][P[j + 1]]` is `true`. A path is a fitting path, defined by `FittingPath(N, G, x, y, P, Q)` in line 184, if in addition to all the above, all elements in the path are in some component set `C`.

To make the verification easier, there also is an existential quantifier for each type. The existential quantifiers `ExPath(N, G, x, y, len)` in line 179 and `ExFittingPath(N, G, x, y, len, C)` in line 190 express that there exists a normal or fitting path `P` from `x` to `y` in `G` with at least length `len`.

4.4.7 Sequence Paths

Sequence paths (listing 9) are a specific kind of path over sequences instead of over an adjacency matrix. They too are represented as a sequence themselves, but they work on our implementation of the union-find data structure. This definition can be used to show that there is a path from some state to another state in the same strongly connected component, which is particularly useful in the proofs of the lemmas in section 4.4.9.

Listing 9: *Sequence paths.*

```
193 // determines whether P is a path from x to y in the sequence S.
194 requires UnionFind(N, S);
195 static pure boolean SeqPath(int N, seq<int> S, int x, int y, seq<int> P) =
196     0 <= x && x < N && 0 <= y && y < N &&
197     0 < |P| && P[0] == x && P[|P| - 1] == y &&
198     (\forall int j; 0 <= j && j < |P|; 0 <= P[j] && P[j] < N) &&
199     (\forall int j; 0 <= j && j < |P| - 1; S[P[j]] == P[j + 1]);
200
201 // existential quantification over paths in S of length at least len.
202 requires UnionFind(N, S);
203 static pure boolean ExSeqPath(int N, seq<int> S, int x, int y, int len) =
204     (\exists seq<int> P; len <= |P|; SeqPath(N, S, x, y, P));
205
206 // defines how a path from x to y is built if it exists.
207 requires UnionFind(N, S);
208 requires ExSeqPath(N, S, x, y, 2);
209 ensures SeqPath(N, S, x, y, \result);
210 ensures |\result| >= 2;
211 static seq<int> getSeqPath(int N, seq<int> S, int x, int y) {
212     if (S[x] == y) {
213         return [x, y];
214     } else {
215         assert (\forall seq<int> P; SeqPath(N, S, x, y, P) && |P| >= 2;
                SeqPath(N, S, S[x], y, tail(P)));
216         // Explicit assert needed (to prove ExSeqPath(N, S, S[x], y, 2))
217         return [x] + getSeqPath(N, S, S[x], y);
218     }
219 }
```

The definition of a sequence path is given, along with an existential quantifier over that path. Additionally, we also need a function that defines how a path from one state to another is defined, given as a precondition that it does exist. All three functions also require the sequence to be a valid union-find with respect to `N` as a precondition.

`SeqPath(N, S, x, y, P)` in line 195 defines a valid sequence path, where `P` is a valid path in `S` iff `x` and `y` are valid with respect to `N`, the length of the path is at least 1, the first element in the path is `x`, the last element is `y`, all elements in the path are valid with respect to `N`, and for all successive states `P[j]` and `P[j + 1]` in the path, the value at index P[j] is P[j + 1]. The existential quantifier `ExSeqPath(N, S, x, y, len)` in line 203 specifies that there exists some path `P` with at least length `len` in `S`.

Finally, the method `getSeqPath(N, S, x, y)` in line 211 defines how a path from `x` to `y` would be build if it exists. This is needed to in some cases instantiate a path as soon as we have proven it exists. A path is constructed recursively, and the method ensures that the resulting return value is a path from `x` to `y`, and that the length of the path is greater or equal to 2.

4.4.8 SCCs

All three types of SCC (PSCC, FSCC and regular SCC) are defined in listing 10. The definitions are relatively one-to-one with those from section 2.1.5. For a component `C` to be an `PSCC` (line

Listing 10: *Strongly connected components.*

```
225 // defines a partial SCC
226 requires AdjacencyMatrix(N, G);
227 static pure boolean PSCC(int N, seq<seq<boolean>> G, set<int> C) =
228     (\forall int v; v in C; (\forall int w; w in C && w != v;
                                             ExPath(N, G, v, w, 1) &&
229                                          ExPath(N, G, w, v, 1)));
230 // defines a fitting SCC
231 requires AdjacencyMatrix(N, G);
232 static pure boolean FSCC(int N, seq<seq<boolean>> G, set<int> C) =
233     (\forall int v; v in C; (\forall int w; w in C && w != v;
                                             ExFittingPath(N, G, v, w, 1, C) &&
234                                          ExFittingPath(N, G, w, v, 1, C)));
235 // defines a maximal SCC
236 requires AdjacencyMatrix(N, G);
237 static pure boolean SCC(int N, seq<seq<boolean>> G, set<int> C) =
238     FSCC(N, G, C) &&
239     (\forall set<int> Cp; NonEmptyComponent(N, Cp) && FSCC(N, G, Cp) &&
                          Cp != C; !(\forall int i; i in C; i in Cp));
```

227), there needs to exist a path from every state in the component to all other states in the component. An `FSCC` (line 232) is similar, but here the states in the aforementioned paths have to be fitting, i.e. only contain states that are also in the `FSCC`. A regular `SCC` (line 237) is a maximal `FSCC`. This means that there are no other FSCCs that contain this SCC. These definitions are currently unused in the verification, since the correctness of the algorithm is not yet verified.

4.4.9 Lemmas

The program makes use of several lemmas to aid in the verification of two important post-conditions of the `unite(N, S, v, w)` operation on a union-find structure. Verbatim, these postconditions are as follows:

Listing 11: *Postconditions of the `unite` operation.*

```
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));
```

There are five lemmas in total, of which lemma 4 and lemma 5 prove the above postconditions, and the other three are auxiliary to these two lemmas. Each lemma requires the input union-find `S`, and the input state arguments `v`, `w` and/or `r` to be valid with respect to `N`. In the remainder of this section the lemmas are listed and the ultimate purpose of these lemmas is explained, and in appendix B the proofs of these lemmas are given. These proofs closely correspond to the proof in the PVL code.

Lemma 1. Given a union-find sequence S and a state v, then there exists a path $v \to^* S(v)$ in S with at least length 2.

Lemma 2. Given a union-find sequence S and states v and r, where $S_r = r$, and there exists a path $v \to^* r$ in S with at least length 2, then $S(v) = r$.

Lemma 3. Given a union-find sequence S, a state v and a path $P = v \to^* S(v)$ in S with $|P| >= 2$, then $\forall i \in P : S(i) = S(v)$.

Lemma 4. Given a union-find sequence S, two states v and w, and the sequence $T = S_{S(w) \to S(v)}$, then $\forall i \in [0, N) : S(i) = S(w) \implies T(i) = S(v)$.

Lemma 5. Given a union-find sequence S, two states v and w, and the sequence $T = S_{S(w) \to S(v)}$, then $\forall i \in [0, N) : S(i) \neq S(w) \implies T(i) = S(i)$.

As mentioned before, lemmas 4 and 5 directly correspond to the two postconditions of the `unite` operation in listing 11. The intuitive interpretation of the postconditions is that after we unite the partitions of states `v` and `w`, i.e. do `T = unite(N, S, v, w)`, the two following properties hold for the resulting union-find `T`:

1. After the unification of `v` and `w`, all states that previously had the same representative as `w` have the same representative as `v`.
2. The representatives of all states that previously did not have the same representative as `w` remain unchanged after the unification of `v` and `w`.

In more general terms, these two postconditions restrict the changes that the `unite` operation makes to the union-find structure. This restriction on the operation in turn maintains an important invariant of the algorithm, which is explained in more detail in the next section.

4.4.10 Algorithm

Since the code for the algorithm is too long to put into a single code snippet, it will be broken up. We will first go over the method contract, starting with the basic invariants, which ensure the validity of the data structures with respect to the graph size `N` and prove memory safety. We continue with the pre- and postconditions, and we complete the method contract with the functional invariants (i.e. invariants that specify the functionality of the algorithm). Finally the loop invariants for the two loops in the algorithm are shown, before explaining the method body.

Method contract

Listing 12: *Basic invariants.*

```
context_everywhere Perm(N, 1\2) ** Perm(G, 1\2) ** Perm (S, 1) ** Perm(R, 1) **
                   Perm(Explored, 1) ** Perm(Visited, 1);
context_everywhere 0 < N && AdjacencyMatrix(N, G) && UnionFind(N, S) && Stack(N, R)
                   && Component(N, Explored) && Component(N, Visited);
```

These two first invariants (listing 12) define read access to `N` and `G`: the size of the graph an the adjacency matrix, resp. Write access is granted to the union-find `S`, the stack `R`, and the sets `Explored` and `Visited`. Finally, we make sure that all the above are valid with respect to `N`.

Listing 13: *Preconditions.*

```
407 requires 0 <= v && v < N;
408 requires !(v in Visited);
409 requires !(v in Explored);
410 requires !(v in R);
```

The preconditions of the algorithm method (listing 13) assume that `v` is valid with respect to `N`, and `v` is not in either `Visited`, `Explored`, or on `R`. These last three preconditions over `v` indicate that the graph search only visits states it has not visited before, i.e. are not in any of the exploration-state sets or on the stack.

Listing 14: *Postconditions.*

```
ensures N == \old(N);
ensures (\forall int x; x in \old(Visited); x in Visited);
ensures (\forall int x; x in \old(Explored); x in Explored);
```

The postconditions in the method contract (listing 14) ensure that after exiting the function `N` is unchanged. Additionally, `Visited` and `Explored` should be strictly increasing: all elements that were previously in the sets are still in them (though there can be more).

These invariants, preconditions and postconditions express properties about the data used in the algorithm, but they do not prove any property of the functionality of the algorithm. The next listing describes the functional invariants.

Listing 15: *Functional invariants.*

```
417 context_everywhere (\forall int x; x in R; x in Visited);
418 context_everywhere (\forall int x; x in Unseen(); part(N, S, x) == {x});
419 context_everywhere (\forall int x; x in R; x == rep(N, S, x));
420 context_everywhere (\forall int x; x in Live(); rep(N, S, x) in R);
```

The important functional invariants are shown in listing 15. The invariant in line 417 specifies that `R` is a subset of `Visited`. It makes sure that the precondition in line 410 is met when the method is called recursively.

The second invariant, in line 418, helps maintain the invariant in line 419: when a new state `v` gets added to the stack (line 432), we know that since `v` was previously unseen (by line 408), the partition of `v` is still {`v`} and so the representative of `v` is itself when we enter the loop in line 442.

The last two invariants verify equations 4.1 and 4.2 from section 4.3. More specifically, the invariants in lines 419 and 420 combined preserve these equations as follows. For equation 4.1 we can show that all partitions for the states in `R` are disjoint and together contain all `Live` states. All partitions are disjoint because a partitions cannot have two representatives, and by the invariant in line 419 `R` only contains representatives. The partitions of the representatives on `R` contain all `Live` states because by the invariant in line 420 the representatives of all `Live` states are in `R`. Equation 4.2 also follows from this last invariant, since representatives are unique we can say that all `Live` states have a unique representative on `R` (namely its representative).

Loop invariants

Listing 16: *Loop invariants.*

```
437 loop_invariant 0 <= v && v < N;
438 loop_invariant 0 <= w && w <= N;
439 loop_invariant N == \old(N);
440 loop_invariant (\forall int x; x in \old(Visited); x in Visited);
441 loop_invariant (\forall int x; x in \old(Explored); x in Explored);
```

There are a few basic loop invariants (listing 16), that are present for both loops in the algorithm. They describe the bounds on `v` and `w`, and establish the three postconditions of the algorithm method that were described earlier in this section. Note that the above loop invariants are for the outer loop in line 442. The invariants for the inner loop in line 458 are almost the same, however for the constraints on `w` we have `w < N` instead of `w <= N`.

Method body

The algorithm is implemented in the `SetBased` method. It closely corresponds to the pseudocode in algorithm 1 in section 4.1.2. There are some small differences, but these will be explained when they are encountered. The PVL code of the algorithm implementation can be found in listing 17.

Method declaration

The method is declared in line 429, and takes a state `int v` as its argument. This state is immediately put in `Visited` and pushed onto `R`, simlar to the pseudocode.

Outer loop: DFS search

The loop over all successors of `w`, represented by **forall** $w \in \mathsf{Succ}(v)$ **do** in the pseudocode, is implemented by a while loop in line 442. With `w` initialised to `0`, we loop over all possible states `w` and check in line 443 if `w` is a successor of `v`, i.e. `G[v][w]`. At the end of the loop, in line 470, we increment `w`.

The condition that if `w` is a successor of `v` (pseudocode: **if** $w \in \mathsf{Explored}$ **then continue**) is implemented by an if-statement with the condition `!(w in Explored)`, in line 444. The next two conditions in the algorithm are implemented the same as in the pseudocode.

If `!(w in Visited)`, i.e. `w` is not yet visited by the DFS, we recursively call `SetBased(w)`, and continue the search. At this point, we need to make sure all preconditions and invariants of `SetBased` are established. For the functional invariants this is guaranteed by the fact that they are `context_everywhere`, and so also interpreted as loop invariants. For the preconditions we require that `w` is not in `Visited`, `Explored` or `R`, which is guaranteed by the invariant in line 417 and the two conditions in lines 444 and 445. The bounds on `w` with respect to `N` are guaranteed by one of the basic loop invariants and the while-condition.

Listing 17: *Algorithm implementation in PVL.*

```
void SetBased(int v) {

    Visited = Visited + set<int> {v};
    R = push(N, R, v);

    int w = 0;

    // --- basic loop_invariant -- \\
    [...]
    while (w < N) {
        if (G[v][w]) {
            if (!(w in Explored)) {
                if (!(w in Visited)) {

                    SetBased(w);

                } else {

                    // --- basic loop_invariant -- \\
                    [...]
                    loop_invariant w in Live();
                    while (part(N, S, v) != part(N, S, w)) {

                        assume v in Live();  // Assumption [...]

                        tuple<seq<int>, int> t = pop(N, R); R = getFst(t); int r = getSnd(t);

                        S = unite(N, S, top(N, R), r);

                    }
                }
            }
        }
        w++;
    }

    if (|R| > 0) {  // precondition for top [...]
        if (v == top(N, R)) {

            Explored = Explored + part(N, S, v);
            assert (\forall int x; x in Live(); rep(N, S, x) in R);  // Explicit assert needed
            tuple<seq<int>, int> t = pop(N, R); R = getFst(t);

        }
    }
}
```

Inner loop: collapse partitions

The following happens if we reach the else-part starting in line 449, so when `w` is already in `Visited`. Here we enter the second while loop, where we collapse the partitions of the top two states of `R` so long as the partitions of those states are not equal: `while (part(N, S, v) != part(N, S, w)) {[...]}`. Besides the basic loop invariants from before, we add one extra in line 457 which states that `w` will remain in `Live`. Initially, `w` is in live because of the two conditions in lines 444 and 445. `Live` is not modified within the body of the while loop, so the loop invariant is maintained. Ideally, we would also want `v` to be in `Live`. However, to do this we would need a loop invariant `v in Live()` on the both the outer and inner loops, and so we would ultimately need a new postcondition for `SetBased` that states that `Live` is strictly increasing. However, we were not able to verify this in this thesis, and so in line 460 we assume that `v` is in `Live`. This is the only assumption we make, and it would be the logical next step in the verification if it is continued (more on this in section 4.4.11).

Now that we know that both `v` and `w` are in `Live`, we know there are at least two states on `R` (by the invariant in line 420) we can pop the top state of `R` in line 462 and unite the partitions of the popped state and the new top state in line 464. The `unite` operation ensures that the resulting sequence is a valid union-find, and that lemmas 4 and 5 hold. These last two lemmas help maintain the invariant in line 420, which in turn is essential to the correctness proof of the algorithm by verifying the two important properties form 4.3. The lemmas state that in this case, only the representatives of the states that previously had `r` as their representatives have changed to the new top state on `R` (and nothing else has changed). This means that these states still have their representative on `R`.

Marking states explored

Finally, after all successors are visited, we can potentially mark states as completely explored (and implicitly report an SCC). We only need to mark states explored if `v` (the state we are currently visiting) is the top state on `R`. In line 473 we make sure we satisfy the precondition of `top` (if `|R| <= 0` then `v == top(N, R)` never holds, so this is valid). In the next line we check if `v == top(N, R)`, and if this is the case we add the partition of `v` to `Explored` in line 476. In order to maintain the invariant in line 420, we need to help the verifier by explicitly asserting the invariant, and lastly we pop `v` in line 478 and maintain the other important invariant in line 419.

4.4.11 Remaining work

As mentioned earlier, the verification of the set-based SCC algorithm is not complete. The final part of the correctness proof outlined in section 4.3 has still to be completed. In the previous section it was revealed that proving the invariant `v in Live()` (for both the outer and inner loops) is the most logical next step. Proving this invariant will remove the need of the assumption in line 460. Here we will suggest how this can be done, and which steps could be taken after that.

Proving the loop invariant for the inner loop is simple, since the `Live` set does not change in the loop body. The tricky part is proving the invariant for the outer loop. In the outer loop body, there is a possibility that `SetBased` gets called. This means that we need to add a postcondition to the method contract (and potentially some supporting loop invariants to both loops) that states that `Live` is strictly increasing, i.e. if some state is in `Live` at the start of the method, then it is also in `Live` at the end of the method.

We already specified that `Visited` and `Explored` are strictly increasing, but this does not guarantee that `Live` is too. A potential method to prove the latter is to keep track of which states get added to `Explored` (and so removed from `Live`) at the end of the method, and making sure that these states were not in `Live` at the beginning of the method.

After the assumption in line 460 is proven, what remains is to use the functional invariants in the method contract to prove postconditions describing the soundness and correctness of the algorithm. This may require additional invariants to be formulated (such as the currently unused invariant over the union-find, from section 4.4.3), and new lemmas may need to be proven. An example of such a postcondition could be that for all states in `Explored`, the partition the state is in is a maximal SCC. Together with the fact that after the algorithm has finished, all states are in `Explored` (which may also need to be verified), this would guarantee soundness and correctness.

A last possible addition could be to explicitly formalise the invariants from section 4.3, instead of having them implicitly follow from the functional invariants of the method. This could be done by adding them as invariants, or by adding assertions in the relevant places. This would benefit the reader of the code by making clear what invariants hold at which point in the algorithm, and it might also help VerCors in the verification process.

5 VERIFICATION OF GRAPH ALGORITHMS USING VERCORS

In order to answer RQ2: *How can the VerCors tool set be improved for the verification of other (parallel) graph algorithms*, we defined two sub-questions, which will be answered in this chapter.

The first question, *How suitable is VerCors for the verification of the model checking algorithm from a user perspective*, will be answered by analysing the verification of the set-based SCC algorithm from the perspective of a VerCors user, and by looking at the user experience with VerCors in section 5.1. To answer the second sub-question, *What can be learned from other verification techniques and tools to improve the verification of graph algorithms with VerCors*, we then explore several related efforts in the fields of verification using both interactive and automated theorem provers in section 5.2. Finally, we use the answers to these questions to suggest improvements to the VerCors tool set in section 5.3.

5.1 Verification of set-based SCC

This section looks at the verification of the set-based SCC algorithm in chapter 4, and verification of graph algorithms in general, from the perspective of a VerCors user. The goal is to identify the advantages and disadvantages of using VerCors for the verification of a high-level graph algorithm.

5.1.1 Usability of previous work

The verification in chapter 4 is loosely based on the previous work of Oortwijn, who verified another model checking algorihtm called *PNFDS* (Parallel Nested DFS) [Oor19] using VerCors. This means that there were several parts of this work that could be reused, notably the graph-related concepts in PVL. Additionally, the manual proof outlined by Bloemen [Blo19] was used to guide the verification. Below the influences of these two previous works are explored.

Graph-related concepts

There were several important concepts that can be reused from Oortwijn's verification of the PNDFS algorithm:

1. The specification of the graph as adjacency matrix, i.e. what is a valid adjacency matrix with respect to the graph size. The fact that the graph is implemented as a matrix influences several other parts of the verification, such as path definitions and successor selection.
2. The specification of valid paths in the graph, and existential quantifiers over such paths.
3. The global state. Though the global state contains different elements when compared to the PNDFS verification, the concept of having the graph and several data structures such as the union-find and stack as class attributes instead of as argument to the algorithm method was useful.

These points have in common that they are not specific for any one algorithm. This means that they could be used for even more graph algorithms. However, there are also many concepts that are more specialised for use by certain algorithms. For example, the union-find data structure used in the SCC algorithm had to be formalised and verified from scratch. This shows that there is potential for reuse of at least some concepts previously used in related work.

Manual proof

The proof outline of the algorithm given by Bloemen provides an idea of how to structure the verification in VerCors. The proof is not complete, but it provides two key invariants that can be formalised and used (see sections 4.3 and 4.4.10).

Though this proof outlines the steps that were needed to prove the correctness of the algorithm, there are two challenges with using a pen-and-paper proof as a basis for verifying an algorithm using deductive verification in the manner VerCors does.

Firstly, the proof is generally more abstract, and does not concern itself with implementation details that could complicate, or even invalidate the approach taken in the proof. In the case of the set-based SCC algorithm, this effect is relatively minor since the formalisation of the two invariants is something that can be done in VerCors by annotating the method and adding loop invariants. However, even there the formalisations of the two invariants do not map one-to-one to the invariants in the proof outline, as can be seen in section 4.4.10.

Secondly, the proof might implicitly assume that certain parts of the algorithm are already proven correct, such as data structures or representations. An example of this is that the proof outline of the set-based SCC algorithm did not concern itself with the correctness of the operations on the stack or union-find data structures. In this case, verifying the union-find structure is not trivial, and the verification of this data structure is closely linked to the two invariants of the proof outline (see section 4.4.9).

5.1.2 Features

VerCors has some features that can impact the verification process, of which the most important are the supported languages, triggers, and most importantly concurrency support. Below we explain the impact of these features on the use of VerCors in general, and on the verification effort in this thesis.

Languages

VerCors supports, besides the prototyping language PVL, several mainstream programming languages. As a result, the tool set can be used to verify very concrete and executable implementations of programs, and there is no need for abstraction or omission of implementation details.

On the one hand, this can be very beneficial: if we manage to verify the implementation of the program, there are no other steps left and the program can be used or published immediately. On the other hand, implementation details often introduce more complexity to the verification, and so it might be difficult to verify the program. If these implementation details were left out, it can be easier to verify the concepts behind the program. If the latter is the goal, leaving out the implementation would be preferable.

In this thesis we use PVL to verify the algorithm, meaning that the code cannot be executed and so it might not be considered fully implemented. However, it would not be difficult to translate the PVL code to another object oriented language, e.g. Java, which gives a full implementation.

Triggers

Triggers (or patterns) are employed by SMT solvers, which is the type of solver used by the VerCors back end Viper. In short, they are candidates to instantiate universal quantifiers, and they help the solver prove these quantifiers. Important for triggers as a feature is that these triggers can be either heuristically determined or provided by the user.

VerCors allows for the user to explicitly provide triggers in quantifiers, which gives the user more control over the verification process. However, understanding how the manual selection of triggers impacts the verification, and to employ this in a meaningful and beneficial way is not trivial and requires a understanding of the underlying technology. This means that as a feature, it is most useful for experienced users with knowledge of the VerCors architecture.

The verification of the set-based SCC algorithm in this thesis uses explicit triggers to help reduce the verification time: by directing the underlying solver, we can make sure it tries out less dead ends in its search for a solution.

Concurrency

An important feature of the VerCors tool-set is its ability to verify the functional correctness of concurrent programs, by employing Concurrent Separation Logic. This feature does add complexity to the verification of any program, even sequential programs since fractional permission for all data needs to be defined. However, for sequential programs this added complexity can often remain quite limited, and the benefit of being able to also verify concurrent programs justifies this feature. There are however some cases where these permissions do complicate verification of a sequential program, for example if the graph in the set-based SCC algorithm was represented by nodes pointing to other nodes. In that case, the permissions needed at any point in the algorithm become much more complex.

A benefit of having the same environment for both sequential and concurrent programs is that, at least in the case of graph algorithms, the verification of a concurrent version of an algorithm can be preceded by the verification of the sequential version. For example, there exists a concurrent version of set-based SCC algorithm [BBDL+18], for which the verification of the sequential version can serve as a basis.

5.1.3 User experience

Analysing the user experience when using VerCors is an important part of investigating the usability of the tool set from a user perspective. We look at four important parts of the user experience: documentation, the development environment, the efficiency of the tool set, and the error feedback it provides.

Documentation

The documentation of the VerCors tool set is extensive, and contains information on installation and setup, IDE support, most verification concepts (e.g. ghost code, permissions, parallel structures, inheritance etc.), common error messages and syntax. This documentation is readily available online[1], and is useful for both starting and experienced users.

Development environment

VerCors supports development of programs and their verifications in several popular IDEs, as well as via the command line on all major operating systems. Furthermore, users can try out the tool online, before even downloading anything. This means that tool set is easy to set up on most systems, and users can start developing for or using the tool with relative ease.

Verification time and termination

Important for the user experience when working with any type of tool set, is its usability in terms of time efficiency. With VerCors, the most important factor is the verification time. As the complexity and size of the to-be-verified program grows, so does the verification time of the program. This verification time can greatly impact the efficiency of the work - when it is longer, it is less efficient to quickly and incrementally run new additions to the verification.

There are some remedies that can help with reducing the overall verification time. Firstly, functions or methods that are already completely verified can be made abstract, i.e. without a body or implementation. This means that the verifier assumes the contract of the function or method without verifying it, and so this will cut down on the verification time. One has to be careful to not inadvertently change the contract while the function or method is abstract, as to not invalidate the verification result. Another way to help reduce the verification time is by guiding the verifier using triggers. Here too care must be taken to not guide the verifier down a wrong path, which will increase the verification time or even make it so that the program will not be verified at all.

[1]VerCors documentation: `https://vercors.ewi.utwente.nl/wiki`

In some cases, the verification does not terminate at all. Due to the fact that verification times can be very long, the user will not know this for sure, and VerCors does not give any information about which part of the verification causes this non-termination. This means that it can be difficult to find the cause and to remedy the situation.

The verification of the set-based SCC algorithm in this thesis started to take a long time, in some cases up to 40 minutes. Abstracting functions does help reduce this time, but since the most complex part of the verification was the algorithm itself (which cannot be abstracted) the use was limited. As mentioned previously, triggers are used, but sparingly. In the case of this algorithm, the most useful way to reduce time was to find the invariant or condition that causes the long verification time, and try to reformulate or remove it from the verification. This is not an obvious process, and there are no strict rules for when an invariant or condition causes this long verification time. This adds another layer of difficulty to verifying complex programs using VerCors.

Error feedback

There are several types of feedback the user needs when using VerCors. The first is feedback about the verification. This means that, if the verification fails (i.e. the program could not be verified), the user needs to know what the counterexample is that the verifier found. VerCors does this by showing the line in the code that failed, and sometimes even the specific part of the line. This is usually enough information for the user to find the problem. However, as mentioned previously, when the verifier neither proves or rejects the verification, this feedback is absent and it can become more difficult to know what went wrong.

Syntax errors are another common error the user can expect. These errors can sometimes be somewhat confusing in VerCors, but ultimately they almost always point the user to the location of the error, and with some careful inspection even the most obscure errors can be found.

Lastly, there might be bugs in the VerCors code, which will also get reported to the user. These errors are more cryptic, and if the user does not know how the back-end of VerCors works, it is sometimes nearly impossible to remedy them without changing the verification. In the verification in this thesis, two such bugs were encountered (see section 4.4) and the source code of VerCors needed to be changed. This was only possible with the help of other, more experienced and knowledgeable users. While this can make the user experience worse, it is also not easy to find a quick solution to this. VerCors does allow users to provide feedback and create issues, and users can contact the VerCors team.

5.2 Related verification efforts

In this section related efforts in proving the correctness of graph algorithms are listed. The methods employed by these efforts are compared with the verification method of the set-based SCC algorithm from this thesis. There are two types of approaches that we explore: *interactive proofs* and *automated verification*. We list the differences between these methods and our own effort, and explore the perceived advantages and disadvantages of each method, if applicable, in sections 5.2.1 and 5.2.2.

5.2.1 Interactive proofs

Interactive proofs make use of formalisations of algorithms, and prove this formalisation using an interactive theorem prover like Isabelle/HOL[2] or Coq[3]. This proof is carried out interactively, meaning that the user provides the prover with theorems, lemmas and conjectures, and guidance for the proof method. At the same time, the prover keeps track of all knowledge and proof details, and performs the trivial steps of the proof. In this section, we first look at some examples of model checking algorithm verifications (and other related efforts) that are carried out using interactive provers, before summarising the most important differences between these efforts and the verification in this thesis.

Verification of an LTL model checker in Isabelle/HOL

An executable on-the-fly LTL model checker called Cava was verified in 2013 by Esparza et al. [ELN$^+$13], and further improved in 2018 [BL18]. This verification was done using Isabelle/HOL, more specifically the Isabelle Refinement Framework [Lam16]. The model checker uses a variant of a sequential NDFS (nested DFS) algorithm and a modern SCC-based algorithm, and so that part of the verification is comparable to our own. However, since a full working model checker is verified, other parts of the checker (like the LTL to Büchi automaton translation, see chapter 3) also needed to be verified, which increases the complexity and difficulty greatly.

The method that is used to verify both versions of the model checker has two steps. First, an abstract version of the model checker is verified. This abstract version consists of a few hundred lines of formalised pseudocode. Then, this abstract checker is refined into a real implementation using the Isabelle Refinement Framework. During this second step, abstract mathematical concepts such as sets are replaced by independently verified implementations of these structures. The resulting code can be exported to several mainstream functional languages such as ML, OCaml, Haskell or Scala, and the checker can be used in practice. While it is somewhat slower than other comparable model checkers, there is still great value in having a verified model checker: it can be used to validate other model checkers.

Refinement

An important technique employed in this verification is step refinement, which allows for more modular proofs. An example where the same technique is applied is the verification of the Kruskal algorithm for finding the minimum spanning forest of a graph [HLB19]. This algorithm also uses a union-find like data structure, as well as a graph representation. However, initially these data structures are left abstract: the union-find is modelled by an equivalence relation (where all states in a partition are equivalent), and the graph by a set of edges (and states). Only after the version of the algorithm that uses these abstract data structures is proven correct, the algorithm is refined by changing the abstract data structures to concrete implementations. Important is that these implementations have been independently verified, i.e. they meet the same specification as the abstract versions. This means that the proof of the complete algorithm will not be invalidated after the refinement steps.

VerCors has no built-in tool support for this kind of refinement, and as a result the user cannot easily first verify an abstract version of the algorithm or program, before refining this to an im-

[2]Main page for Isabelle: `https://isabelle.in.tum.de/` (accessed 24-06-2021)
[3]Main page for the Coq Proof Assistant: `https://coq.inria.fr/` (accessed 24-06-2021)

plementation. Instead, the user either has to start by immediately verifying the implementation, or leave parts of the implementation abstract (by using abstract functions) and later implement the abstract parts and verify them.

Imperative framework for DFS algorithms in Isabelle/HOL

Lammich and Neumann provide an imperative framework [LN15] for the verification of sequential DFS algorithms in Isabelle/HOL. This framework is specifically geared towards the development of new, efficient DFS-based algorithms, in an imperative style. It first proves the correctness of an abstract version of the algorithm, then uses the Isabelle Refinement Framework to refine this abstraction to an efficient implementation. This refined implementation of the algorithm can be exported to executable code.

The specificity of the framework means that one general template for DFS-based graph algorithms is given in the form of single while loop. Several invariants are used over this loop to prove this basic template of DFS correct. The framework then allows the user to add *hook functions* to certain extension points in the template to alter the specifics of the algorithm. The algorithm is still correct if these hook functions maintain the already established invariants, and new invariants can be established using the framework that prove additional properties of the algorithm.

The framework is built specifically for DFS algorithms, which means that the application is somewhat limited. The authors state that the design principles of this framework could be used to develop frameworks for other types of algorithms (potentially even concurrent), but these too would have a very specific scope.

A strength of this framework when compared to VerCors is its great potential for the development of new algorithms. Proving correctness for an abstract representation of the algorithm can be easier than proving it for a concrete implementation, since the specifics of the implementation do not have to be taken into account. As long as the refinement process maintains the correctness of the algorithm, and is reasonably automated, it could save time when compared to verifying a concrete implementation.

Relevant for our purposes is that Tarjan's algorithm, one of the precursors to the set-based SCC algorithm, was successfully verified using this framework, illustrating that it could be used in practice. However, even though the framework can be used in practice, it is more intended as a proof of concept to illustrate the design philosophy behind it.

IMP2 framework for imperative code in Isabelle/HOL

The IMP2 framework [LW19] for Isabelle/HOL can reason about programs in a similar way as with deductive verification: the program can be annotated with preconditions, postconditions and loop (in)variants. It uses the same concepts as deductive verification, such as Hoare logic and wp-reasoning. Even though the framework is made to resemble the style of deductive verification, there are some major differences.

Firstly, the algorithms encoded using this framework remain more abstract. Even though the framework uses C-like syntax, the code cannot be executed, and concepts such as lists and sets have no concrete implementation. In contrast, VerCors allows the user to program in several mainstream languages, and the programs can be executable.

A second difference, related to the first, is the scope of the language. The IMP2 framework can be seen as an extension of the IMP language, which comes with the Isabelle distribution. This means that the language used by the framework is a very simple imperative language. Furthermore, the framework has no support for object oriented programs, which might be necessary for certain applications.

The third difference is that the IMP2 framework provides more transparency about the semantics of the language such as wp-reasoning and Hoare logic by using IMP, which explicitly encodes the definitions and rules in Isabelle/HOL. This fact, combined with the simple language, can be great for certain applications, such as in education (which is one of the intended use-cases for the IMP2 framework). VerCors can be considerably less transparent to users, especially if the user is not familiar with the underlying technology.

Finally, the IMP2 framework has no support for concurrent programs, while VerCors' use of concurrent separation logic does allow for this. This too limits the scope and potential for use of the IMP2 framework when compared to VerCors.

Key differences

Summarised, there are a few key differences between verification with interactive provers (notably Isabelle/HOL) and VerCors:

1. The representation of the code in interactive proving is often more high-level and abstract, while in VerCors mainstream languages such as Java and OpenCL are directly used. Some tools allow the generation of functional code in executable languages from the abstract representation, but there is almost no support for imperative and/or object oriented languages. If imperative languages are supported, this mostly is limited to small toy languages.
2. The high level of abstractness used by most interactive methods allows for the omission of implementation details, which can aid in reducing the verification complexity. There are mature techniques to turn a verification of an abstract representation into a verification of an implementation, notably the refinement technique. In VerCors PVL allows for abstract functions, but there is no built-in tool support for refinement.
3. The level of control is lower when using a tool that employs an automatic theorem prover, like VerCors does. Interactive proving, as the name suggest, allows for more interaction with how the prover should solve the problems, and gives the user ample opportunity to manually guide the proof. There are ways to do this in VerCors, notably using explicit triggers in the specification, but this still does not give the user the same level of control.
4. Often the proofs in the interactive provers are more closely related to the pen-and-paper proofs of the algorithms, which usually precede the computer-aided verification. This is mainly due to the different proof style when compared to annotating programs, such as is done in VerCors. Again, this might be partly because of the higher level of abstraction, since pen-and-paper proofs are often abstract as well.

5.2.2 Automated verification

Verification using automated verifiers is another approach to deductive verification. With this approach, the program is modeled in a tool, along with its specification. Typically, this is done by annotating the program code with properties (pre- and postconditions, and (loop) invariants) that can help to prove that the program follows the desired behaviour. This is the approach that VerCors uses, as described in chapter 3. Below, other examples of automated verifiers are explored, along with the most important similarities between those examples and VerCors. The first few tools in this section are more similar to VerCors, but we also look at VeriFast and the Verified Software Toolchain, which are increasingly different. It is useful to look at the latter tools because it shows a perspective further removed from that of VerCors, which can lead to new ideas.

Similar tools

There are several very similar tools to VerCors. These tools have many of the same features as VerCors, and can be used to the same ends. However, most of these tools do not or only partly support verification of concurrent programs. Below for each of the similar tools a short description is given, along with its main selling points.

OpenJML

The *Java Modelling Language* (JML) [LBR99] can be used to write specifications for Java programs. It uses the same style of annotations as VerCors, with pre- and postconditions and (loop) invariants. The basic syntax is almost exactly the same as the syntax from VerCors annotations (see section 3.2.2). JML itself does not include the functionality to verify the program with respect to the written specification, but there are many tools available for this purpose. One of the most used tools is OpenJML [Cok11], which *"provides static analysis, specification, documentation, and runtime checking, an API that is used for other tools, uses Eclipse as an IDE, and can be extended for further research."*. There are efforts to extend open JML for use with multithreaded programs, for example e-OpenJML [KHBZ15].

KeY Project

The KeY Project [ABB$^+$16] is another tool that allows users to annotate Java programs with JML, and functionally verify the (sequential) Java code. Its goal is to integrate formal software analysis methods into the mainstream software development process, and the project was started in 1998. Besides formal verification of Java Code, KeY also facilitates test case generation, a debugging tool, a teaching tool and an Eclipse extension, which all help reach its stated goal.

Why3

Why3 [FP13] is a platform for verification of C and Ada programs. It uses the WhyML language, specifically designed for verification of programs and algorithms. This code can then be verified using several methods, employing among others SMT solvers (like Z3) and interactive provers (such as Coq and Isabelle).

Users can either directly write their programs in WhyML, or use the verifiers FramaC [KKP+15] or SPARK 2014[4] to write C and Ada code, respectively. These two latter verifiers use WhyML as an intermediate language and call provers through the Why3 framework.

Verification of NDFS using Dafny

The Dafny tool was used by van der Pol to verify a model checking algorithm [vdP15], more specifically the *nested depth-first search* algorithm (NDFS). This verification was also inspiration for the proof of the parallel version of NDFS which was carried out by Oortwijn using VerCors [Oor19].

Dafny is a tool very similar to VerCors, in that it allows the user to annotate programs, and uses the Z3 SMT solver to verify the program. Dafny employs a specialised language, not unlike PVL, but it can generate .dll or .exe code for .NET platforms using the code. It has no support for other languages, such as Java or OpenCL. The tool is strictly sequential, but has many of the other features of VerCors such as ghost variables, recursive functions and sets and sequences.

Because the Dafny and VerCors tools are very similar, with the verification of NDFS many of the same obstacles and challenges were encountered as with the verification in this thesis. The first step in both verifications was to account for runtime errors, such as index bounds and permissions. Dafny also checks for termination, and so it has a `decreases` keyword to help with this. VerCors does not have this keyword, and so proving the *total* correctness of a program or algorithm might be easier in Dafny than in VerCors. Proving termination can still be done in VerCors, but it has to be explicitly encoded without the use of the `decreases` keyword. Furthermore, both tools sometimes require explicit witnesses to statements with existential quantifiers, i.e. the user has to explicitly provide an instance that satisfies the quantifier. A last challenge that the two tools share is that it is not always clear what the verifier does or does not know at any point in the verification. A way to discover this is by adding several assertions, and checking if the verifier can prove these.

The VeriFast Program Verifier

VeriFast [JSP+11] is a program verifier that works for (subsets of) the languages C and Java. It allows the user to annotate program code with, among other things, method contracts, which then are used to check that the program does not perform illegal operations and the method contract is satisfied. These annotations are more extensive than those in VerCors, and include separation logic-based method contracts, inductive data types, fixpoint functions and lemma functions. This means that the tool can be described as a hybrid between an interactive and automated verifier, and so the focus of the tool shifts: VeriFast focuses *"more on fast verification, expressive power, and the ability to diagnose errors easily than automation"* [JSP10].

This different focus with respect to verification is the cause for several changes in approach. Firstly, the direct use of separation logic in all annotations means that the user must be much more aware of the concepts behind it. VerCors annotations can contain some aspects of separation logic, such as when specifying permissions, but often assertions are written in "standard" first-order logic.

[4]Main page for SPARK 2014: `https://www.adacore.com/about-spark` (accessed 11-08-2021)

Listing 18: *A lemma and a predicate in PVL (VerCors).*

```
requires [...];  // preconditions
ensures [...];  // postconditions
static void lemma_7_uf(int N, seq<int> S, seq<int> T, int v, int w) {
    [...]  // body
}

static inline pure boolean Stack(int N, seq<int> R) =
    [...];  // definition of predicate
```

Listing 19: *A lemma and a predicate in VeriFast.*

```
lemma void 7_uf(int N, list<int> S, list<int> T, int v, int w)
    requires [...];  // preconditions
    ensures [...];  // postconditions
{
    [...]  // body
}

predicate Stack(int N, list<int> R) =
    [...];  // definition of predicate
```

Another difference is the approach to predicates and lemmas. Where in VeriFast, these are explicitly part of the annotations of the program, in VerCors they are often encoded as (pure) functions or methods, and thus part of the code. Overall, VerCors annotations are more related to the program code, where VeriFast annotations are more expressive without much need for program code beyond the normal implementation. These features could have been used in the verification in this thesis to separate the proofs of the five lemmas (see section 4.4.9 and appendix B) from the program code - currently this distinction is not explicit. Lemma 1 and the `Stack` predicate from the verification in chapter 4 is shown as PVL functions and a VeriFast lemma and predicate in listings 18 and 19, respectively. The difference in syntax between the two examples is minimal, and their usage is the same, though in one example the lemma and predicate are part of the program code and the in the other they are explicitly encoded by a lemma/predicate annotation.

Verified Software Toolchain

The *Verified Software Toolchain* (VST) [App11] is a toolchain that can proof the functional correctness of C programs. It uses its own program logic, called *Verifiable C* (a variant of separation logic) to check proofs of the functional correctness of the C programs. This separation logic is proven sound by Coq. The toolchain also guarantees the correctness of the compiled assembly code. It does this by employing the *CompCert verified C compiler*[5], which is a verified C compiler that can generate efficient code for most platforms. Every tool in the toolchain is machine checked, guaranteeing that the tools can be trusted.

Verifying programs using Verifiable C does not use the same method as VeriFast and VerCors to prove programs correct. Instead of allowing the user to annotate the programs, the VST generates Coq definitions for the program. The user can write the specification of the program

[5]Main page for CompCert: `https://compcert.org/` (accessed 11-08-2021)

in Verifiable C, also using `Coq`. These two parts can then be combined to automatically prove that the functions in the program satisfy the specifications. This means that the way programs are verified with the VST is fundamentally different from VerCors.

The biggest difference between the VST and VerCors when looking at the functionality of the tools is that the VST also guarantees the correctness of the compiled program. Depending on the compiler that is used to compile code verified with VerCors, this might not hold for the programs verified with VerCors. As a result, the VST might be more useful for some applications where this is very important.

Overall, verification with VST has more in common with the interactive verification methods from section 5.2.1 than VerCors has, and so it shares many of the differences with VerCors as those methods. For example, verification with VST might give the user more control over the proof strategies, though not necessarily to the extend of the interactive methods, since there remains an element of automation.

Key differences

There are some key differences between deductive verification using other automated verifiers when compared to VerCors:

1. VerCors is one of the few verifiers that uses CSL to allow for the automated verification of programs written in mainstream languages. Other tools for concurrent programs exist, though most tools work for sequential programs only.
   One of the tools that can verify concurrent programs is VeriFast, though the difference in focus between it and VerCors makes a direct comparison more difficult. VeriFast makes much more direct use of separation logic, and supports more types of annotations.
   Another tool that can verify concurrent programs is the Verified Software toolchain, which uses Verifiable C - a separation logic. This tool differs even more from VerCors than VeriFast, since the way of writing specifications and proving compliance is very different.
2. Other verification tools might give varying levels of control over the verification strategies. VeriFast allows the user to explicitely specify predicates and lemmas as annotations, and the VST allows for much the same control as some techniques using interactive provers. The balance between automation and control is different between these tools, though one is not necessarily better than the other in this regard.

5.3 Suggested improvements

This section will use the findings in the previous two sections (5.1 and 5.2) to suggest several improvements. These consist of general improvements and improvements related to refinement techniques from imperative provers.

5.3.1 General points of improvement

There are a few general potential improvements to VerCors, which mostly come from the user experience in section 5.1.3 and from the related work with automated verifiers in 5.2.2.

Syntax errors

While the current syntax errors of VerCors almost always contain enough information about the error, it is often not immediately clear from the way this information is presented. A relatively simple, but effective improvement to the user experience with VerCors could be to reformat these error messages, and so make them more readable and easier to understand.

Verification time feedback

When using VerCors, the user can choose to log the work the VerCors tool set does to transform the program to be used by Viper (the back end tool used by VerCors). This work mainly consists of parsing the code, and performing passes on the COL AST (see section 3.2). However, as soon as Viper starts the verification process there is no more feedback given to the user. By changing this, and continuing to provide progress status feedback (where possible), the user experience can be improved: no longer would the user have to guess how long the verification will approximately take, and non-termination (or very long verification times) could be spotted early on.

Sequential program verification

There are some instances where the fact that VerCors supports Concurrent Separation Logic (CSL) has an impact on its ability to verify sequential programs. In these cases, the fact that fractional permissions have to be specified (unnecessarily) can increase the complexity of the verification. A possible solution, though perhaps a somewhat drastic, could be to add a separate mode for sequential verification, where all concurrent features of VerCors are disabled. This would remove one of the key strengths and advantages of VerCors however, and so it might not be desirable for that reason.

Guidance of proofs

One of the strengths of an automated verifier like VerCors is that it needs very little guidance for the user. However, sometimes it would be useful to still have the option to guide the verifier. Currently, the user can provide directives by explicitly marking triggers and providing helpful assertions. From section 5.2.1 it is apparent that verification methods using interactive provers provide more guidance, but also less automation. The amount of guidance that VerCors can provide is mainly dependent on the back end, which currently is Viper. If another back end would be (partly) supported, for example an interactive prover, VerCors could give the user more opportunity to guide the verification.

Proof of termination

Many automated verifiers make use of the `decreases` to help prove termination of a program or algorithm. VerCors does not support this, so the proof of termination can sometimes be more difficult. Though adding this keyword can help, it might also mean that proof of termination will be required to verify the program, which is not necessarily desirable.

Explicit lemma and predicate functions.

VeriFast provides functionality to annotate programs with lemma and predicate functions. These functions are not a part of the program code, but can still be used to prove lemmas or define predicates. Such an explicit distinction between auxiliary code and program code could help improve the readability and structure of verifications in VerCors. An example use-case would be the proofs of the five lemmas in this thesis (section 4.4.9 and appendix B).

5.3.2 Refinement and abstraction

From section 5.1, we derive several problems that all suggest a similar improvement point for VerCors. Firstly, several important graph related concepts could be reused from previous work (like [Oor19]). However, new concepts, such as the union-find data structure, needed to be verified from scratch. Additionally, the pen-and-paper proof that the verification might be based on is often more abstract, leading to a mismatch between that proof and the verification strategy. These pen-and-paper proofs might also not be complete. For example, it might consider used data structures and graph representations already proven, but in the verification this is not yet the case, leading to additional and possibly unexpected work.

A solution to these problems could come in the form of using refinement techniques in combination with VerCors. Recall from 5.2.1 that refinement entails the following: take a more abstract version of, for example, a data structure, and show that an implementation of that data structure refines the abstract version, i.e. it follows the same specification. After that, we can substitute the abstract version for the implementation without invalidating the verification of the complete program. These refinement techniques have already been successfully used [ELN$^+$13, BL18, LN15, HLB19, Lam19] (see also section 5.2.1) to verify similar algorithms, though using an interactive prover instead of an automated theorem prover. This does not necessarily mean that tool support for this method cannot be implemented in VerCors, and in the remainder of this section we show an example of the usefulness of this method in VerCors. We then look at the current possibilities of refinement in VerCors, before suggesting how the tool set could support this feature even better.

Example: the union-find

The verification of the set-based SCC algorithm in this thesis provides an example use-case for refinement techniques. A large part of the verification effort was proving the correctness of the union-find structure. It would be ideal if there was the built-in possibility of initially leaving the union-find abstract, and verifying the algorithm using this abstraction. Only after this the the union-find would have to be implemented, and this implementation would have to be shown to refine the abstract version of the union-find used for the initial verification.

A related example of this kind of refinement can be found in the work done by Haslbeck et al. [HLB19], who used refinement techniques in their verification of a version of Kruskal's algorithm [Kru56][6], and in work by Lammich [Lam19], where an imperative version of a union-find is verified using refinement. In both cases, the union-find is initially represented by an abstract equivalence relation, i.e. two states are equivalent to each other in this relation iff they are in the same partition of the union-find. Only in later stages of the verification is the relation refined into an efficient implementation of a union-find.

Doing something similar for the verification in this thesis would accomplish several things. Firstly, it would modularise the verification: the verification of the algorithm and the union-find could be done separately, without invalidating either of them. Secondly, this would make the

[6]Kruskal's algorithm finds the minimum spanning forest of a graph using a union-find.

verified union-find implementation readily available for use in other verification efforts, since its verification is independent of the context. Thirdly, the verification could follow pen-and-paper proofs more closely, since a higher level of abstraction is possible. Lastly, it would make sure that the verification of the algorithm stays focused on its goal of proving the algorithm correct.

Current support for refinement in VerCors

Currently VerCors has some features, most notably in PVL, that can facilitate the type of verification described above, although in a more limited form. Abstract functions or methods are functions or methods declarations without a body, i.e. implementation specified. They can and often do have method contracts, indicating what the pre- and postconditions of the methods are. Potentially, interfaces could be used in the same way as these abstract functions in other languages such as Java, though below we focus on the use of PVL.

In the case of the union-find example from the previous section, this could be used to initially leave the implementations of the operations (e.g. the *find* and *unite* operations) abstract, or unimplemented. The union-find itself could then be abstract as well, for example an equivalence relation. Only later, when the program is verified with this abstract version of the union-find, an implementation is written and verified. This means that the previously abstract operations are now implemented, as well as the union-find itself. This approach, based on the work of Lammich [Lam19], is described in more detail in appendix C.

Further support for refinement in VerCors

While the simple method described above might work in this instance, there are several drawbacks to using it, as well as limits on its application. The first drawback is that it is not certain that the abstract methods can be implemented at all, given the pre-specified method contract. If it is discovered that there is no way to satisfy the postconditions when the method is implemented, it might invalidate the complete verification. Another way this can happen is if the implementation introduces new implementation-specific pre- and postconditions, which require the user to change the complete program verification as well.

This method is also limited to *pure* methods and functions, i.e. methods that do not change the program state. If a program containing abstract methods is verified, implementing the methods in such a way that within the method body the program state is changed can completely invalidate the verification.

Using more mature refinement techniques, the impact of these drawbacks can be reduced, and the limits of the current support by VerCors can be removed: the different method of adding an implementation to the verification isolates it from the full verification, making it more independent and modular. Below we suggest what is needed for refinement methods to be better supported by VerCors.

Mindset change

First of all, the user of VerCors has to change their mindset when it comes to verifying using the tool-set. Currently, abstract functions or methods and assumptions in the code are often seen as temporary, or *to be filled in.* When working with refinement, a layer of abstraction gets introduced, especially at the earlier stages of the verification. While the goal of the user might remain the same: verify an implementation of some program or algorithm, the method to reach this goal changes and so too should the user's mindset. Abstract code should not be seen as missing code, but instead as parts of the verification that will get verified independently, and then integrated into the full verification.

Modules

Ideally independently verified concepts such as data structures are used in the verification of different programs, without the need of copying or rewriting code. A system of modules (or some type of library) containing fully verified programs is a good way to manage this. VerCors would need to allow the user to import these modules in a simple way, for example an import statement. The user can then use the verified code in those modules in their own file, not unlike many mainstream languages.

The contents of these modules should be fully verified, but it can contain both abstract and implemented programs. Both have a different use-case. Abstract modules can be used when verifying an implementation of some or all of the concepts in the module. For example, if the module contains the abstract specification for a union-find and its operations, it can be used to verify an implementation by showing that the implementation refines the abstraction. Alternatively, if the module contains a fully verified program, the contents of this module can be used in another verification. An important benefit over simply including the full code of the module in the new verification - besides the decrease in work - is that the contents of the module do not have to be verified each time the verifier is run, reducing the verification time by a significant margin.

Besides abstract or implemented data-structures or representations, modules can also contain proven properties of these concepts. Examples of this could be the absence of cycles in a tree graph, or the invariant over the union-find from section 4.4.3.

Native support for refinement methods

The Isabelle Refinement Framework (IRF), which is used by most examples of refinement with interactive provers, uses some advanced concepts to facilitate more mature refinement methods. These methods solve the shortcomings of the simple approach that is currently possible in VerCors. For this reason it might be desirable to implement these concepts in VerCors as well, though this is by no means trivial, and ideas for implementation details could be explored in future work.[7] For a description of the IRF applied to verification of algorithms, and of how it addresses the drawbacks of the simple verification method, we refer to [Lam19].

5.3.3 Summary

After analysis of the user experience of verifying a graph algorithm with VerCors, based on the work from chapter 4, and the exploration of related work we suggest several improvements to VerCors. While the user experience of working with VerCors is generally good, especially the feedback given to the user - both in terms of error and progress feedback - could be improved upon. Additionally, the high degree of automation limits the guidance the user can give, which could also be improved. Other tools might give inspiration for potential solutions for these

[7]Due to time constraints it is not feasible to explore these mature methods in combination with VerCors in this thesis.

problems. Notably, refinement and abstraction techniques employed by other verifiers can help streamline the verification process in VerCors, by providing the user with the opportunity to reuse and abstract their verifications.

6 CONCLUSION

In the introduction of this thesis we posed two main research questions, the first of which is RQ1: *What techniques are involved with proving the correctness of a high-level model checking algorithm using the VerCors tool set?* In this thesis, we show that deductive verification can be used to verify a significant portion of a sequential model checking algorithm, namely the set-based SCC algorithm. We proved important invariants about the union-find data structure used by the algorithm, an we laid the groundwork for the correctness proof of the algorithm itself. During the verification the following techniques were used:

- reusing concepts and ideas from previous related efforts,
- annotating the algorithm with key assertions and invariants that help prove correctness, based on a pen-and-paper proof outline,
- encoding data structures and defining predicates and operations over those data structures,
- defining lemmas and their proofs using imperative functions,
- guiding the verifier by specifying explicit triggers and assertions.

Using these techniques gives insight into the process of using VerCors to verify a graph algorithm. We use this insight to help answer RQ2: *How can the VerCors tool set be improved for the verification of other (parallel) graph algorithms?* This question has two sub-questions that help answer it. The first sub-question investigates the suitability of the VerCors tool set for the verification of model checking algorithms. By analysing the verification of the set-based SCC algorithm from the perspective of a VerCors user, and by looking at the user experience with VerCors we find that VerCors is a suitable candidate to be used for this kind of verification, especially if the algorithms are concurrent. However, there is always room for improvement, and several insights were gained trough this analysis that contributed to the list of improvements that answers RQ2. The second sub-question asks what features and ideas from other related efforts. We explore several verification efforts using both interactive and automated theorem provers, and from these explorations we find several other points of improvement. The list of potential improvements, answering RQ2, can be summarised as follows:

- *Syntax errors* - The clarity of syntax errors in VerCors could be improved.
- *Verification time feedback* - VerCors could provide feedback to the user about estimated remaining time for the verification.
- *Sequential program verification* - Concurrent features of VerCors could be optional when verifying sequential programs.
- *Guidance of proofs* - VerCors could provide the user with more opportunities to guide the verification.
- *Proof of termination* - The `decreases` keyword could be added to VerCors to aid in proofs of termination.
- *Explicit lemma and predicate functions* - VerCors could allow the user to explicitly define lemmas and predicates as part of the annotations, instead of part of the program code.

- *Refinement and abstraction* - Adding refinement and abstraction techniques to VerCors could allow the user to more efficiently reuse code, and independently verify parts of a program. This could lead to easier and less time-consuming verification and more modular program structures. This improvement would require a mindset change of the VerCors user, a module or library system and ultimately native support for refinement methods.

Future work

There are several topics that could lead to future work that are based on this thesis. The verification of the algorithm in this thesis is not complete, so more work is needed to finish it. This work is described in more detail in section 4.4.11. Furthermore, the idea from appendix C, where a simple refinement method is used in PVL to show how it could be used to verify the union-find data structure, could be explored. This could be done not only for PVL but also for other languages that VerCors supports (such as Java). Finally, the other improvements points listed earlier in this chapter as the answer to RQ2 could all be explored in greater detail, especially the idea of implementing support for mature refinement techniques in VerCors. Some suggestions for the way these improvements could be realised were made in section 5.3.

REFERENCES

[ABB$^+$16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[AHHH15] A. Amighi, Christian Haack, Marieke Huisman, and C. Hurlin. Permission-based separation logic for multi-threaded java programs. *Logical methods in computer science*, 11(1):2, February 2015. eemcs-eprint-25324.

[App11] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[APPB10] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, page 1–11, USA, 2010. IEEE Computer Society.

[BBDL$^+$18] Jiri Barnat, Vincent Bloemen, Alexandre Duret-Lutz, Alfons Laarman, Laure Petrucci, Jaco van de Pol, and Etienne Renault. *Handbook of Parallel Constraint Reasoning*, chapter Parallel Model Checking Algorithms for Linear-Time Temporal Logic, pages 457–507. Springer International Publishing, Cham, 2018.

[BDHO17] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[BL18] Julian Brunner and Peter Lammich. Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reason.*, 60(1):3–21, 2018.

[Blo19] Vincent Bloemen. *Strong Connectivity and Shortest Paths for Checking Models.* PhD thesis, University of Twente, 7 2019.

[BLv16] Vincent Bloemen, Alfons Laarman, and Jan Cornelis van de Pol. Multi-core on-the-fly scc decomposition. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016*, page 8, United States, 3 2016. Association for Computing Machinery (ACM). eemcs-eprint-26873.

[Boy13] John Boyland. *Fractional Permissions*, pages 270–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1):227–270, 2007. Festschrift for John C. Reynolds's 70th birthday.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[Bv16] Vincent Bloemen and Jan Cornelis van de Pol. Multi-core scc-based ltl model checking. In Roderick Bloem and Eli Arbel, editors, *Hardware and Software: Verification and Testing; Proceedings of the 12th International Haifa Verification Conference, HVC 2016*, Lecture Notes in Computer Science, pages 18–33. Springer, 11 2016. 10.1007/978-3-319-49052-6_2.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking.* Springer, 2018.

[Cok11] David R. Cok. Openjml: Jml for java 7 by extending openjdk. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 472–479, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, oct 1992.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall series in automatic computation. Prentice-Hall, 1976.

[ELN+13] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable ltl model checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 463–478, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[ELPvdP12] Sami Evangelista, Alfons Laarman, Laure Petrucci, and Jaco van de Pol. Improved multi-core nested depth-first search. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, pages 269–283, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[EPY11] Sami Evangelista, Laure Petrucci, and Samir Youcef. Parallel nested depth-first searches for ltl model checking. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, pages 381–396, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[HLB19] Maximilian P. L. Haslbeck, P. Lammich, and Julian Biendarra. Kruskal's algorithm for minimum spanning forest. *Arch. Formal Proofs*, 2019, 2019.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hol12] Gerard J. Holzmann. Parallelizing the spin model checker. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, pages 155–171, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[JOSH18] Sebastiaan J. C. Joosten, Wytse Oortwijn, Mohsen Safari, and Marieke Huisman. An exercise in verifying sequential programs with vercors. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, page 40–45, New York, NY, USA, 2018. Association for Computing Machinery.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 304–311, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[JSP$^+$11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[KHBZ15] Jorne Kandziora, Marieke Huisman, Christoph Bockisch, and M. Zaharieva. Runtime assertion checking of jml annotations in multithreaded applications with e-openjml. In R. Monahan, editor, *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs (FTfJP 2015)*, page 8, United States, July 2015. Association for Computing Machinery (ACM). 10.1145/2786536.2786541 ; null ; Conference date: 07-07-2015 Through 07-07-2015.

[KKP$^+$15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[Lam16] Peter Lammich. The imperative refinement framework. *Archive of Formal Proofs*, August 2016. `https://isa-afp.org/entries/Refine_Imperative_HOL.html`, Formal proof development.

[Lam19] Peter Lammich. Refinement to imperative hol. *J. Autom. Reason.*, 62(4):481–503, April 2019.

[LBR99] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: A notation for detailed design. In *behavioral specifications of Businesses and Systems*, pages 175–188. Springer, 1999.

[LLvdP$^+$11] Alfons Laarman, Rom Langerak, Jaco van de Pol, Michael Weber, and Anton Wijs. Multi-core nested depth-first search. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, pages 321–335, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[LN15] Peter Lammich and René Neumann. A framework for verifying depth-first search algorithms. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, page 137–146, New York, NY, USA, 2015. Association for Computing Machinery.

[Low15] Gavin Lowe. Concurrent depth-first search algorithms based on tarjan's algorithm. *International Journal on Software Tools for Technology Transfer*, 18, 05 2015.

[LvW10] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Formal Methods in Computer Aided Design*, pages 247–255, 2010.

[LvW11] Alfons Laarman, Jan Cornelis van de Pol, and M. Weber. Parallel recursive state compression for free. In *Proceedings of the 18th International SPIN Workshop, SPIN 2011, Snow Bird, Utah.* ArXiv, 4 2011.

[LW19] Peter Lammich and Simon Wimmer. IMP2 - simple program verification in isabelle/hol. *Arch. Formal Proofs*, 2019, 2019.

[Mun71] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, Dec 1976.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1):271–307, 2007. Festschrift for John C. Reynolds's 70th birthday.

[Oor19] Wytse Hendrikus Marinus Oortwijn. *Deductive techniques for model-based concurrency verification.* PhD thesis, University of Twente, Netherlands, 12 2019.

[RDLKP13] Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. Three scc-based emptiness checks for generalized büchi automata. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 668–682, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[RDLKP15] Etienne Renault, Alexandre Duret-Lutz, Fabrice Kordon, and Denis Poitrenaud. Parallel explicit model checking for generalized büchi automata. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 613–627, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[RDLKP16] Etienne Renault, Alexandre Duret-Lutz, F. Kordon, and Denis Poitrenaud. Variations on parallel explicit emptiness checks for generalized büchi automata. *International Journal on Software Tools for Technology Transfer*, 19, 04 2016.

[SE05] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–190, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[TvL84] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984.

[VBBB09] Kees Verstoep, Henri Bal, Jiri Barnat, and Luboš Brim. Efficient large-scale model checking. *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 05 2009.

[vdP15] Jaco C. van de Pol. Automated verification of nested dfs. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems*, pages 181–197, Cham, 2015. Springer International Publishing.

A LIST OF MODEL CHECKING ALGORITHMS

Following is a list of several algorithms mainly discussed in [BBD+18] and [Blo19]. This list is compiled in order to explore the different algorithms that could be verified in the project. Only one of these algorithms is used in this paper, but they are nonetheless important to illustrate both the similarities and difference between the various algorithms, and to get an idea of what the landscape regarding these algorithms looks like.

Name Depth-First Search (DFS)
Type Sequential Depth-First Search for Accepting Cycles
Description This is a simple DFS algorithm, where eventually all states of an automaton are explored. The search order is such, that children are favoured over siblings when choosing the next node to explore – as supposed to Breadth-First Search (BFS) which favours siblings. The running time is generally linear over the number of states plus the number of transitions for both DFS and BFS, but since DFS can easily be used *on the fly* to detect cycles (generating the states as they are needed) this is most often used. A simple example is shown in [BBDL$^+$18] (Algorithm 12.2), which works on weak TGBAs (TGBAs for which in any of its SCCs all transitions have the same accepting mark) and SBAs.

Name Nested Depth-First Search (NDFS) [CVWY92]
Type Sequential Depth-First Search for Accepting Cycles
Description This is an alternative version of the DFS search algorithm, but specifically for a SBA (with only one acceptance mark). This algorithm also runs in linear time with respect to the graph size. Basically, there are two separate searches: the blue search and the red search, where the blue search traverses the graph in DFS order, and if it encounters an accepting state, the red search – also DFS – is started from this state. If this red search then encounters a cyan (partially explored in the blue search) state it means that it can reach itself (this is a property of DFS) and so an accepting cycle is reported. A version on this algorithm can be found in [BBDL$^+$18] (Algorithm 12.3). The shown algorithm is an optimised version, derived from the work of Schwoon and Esparza [SE05].

Name Strongly Connected Component based (SCC-based) [RDLKP13]
Type Sequential Depth-First Search for Strongly Connected Components
Description Instead of looking for accepting cycles, like the two previous algorithms, this algorithm instead computes all SCCs of a generalised Büchi automaton. This algorithm is still based on the DFS search principle, but does not need a nested search and also runs in linear time. These kinds of algorithms are based on [Tar72], who proposed an algorithm (Tarjan's algorithm) that partitions the set of states into SCCs. The algorithm in [BBDL$^+$18] (Algorithm 12.4) uses the principles of Tarjan's algorithm, along with a union-find database structure as an emptiness check for TGBAs. union-find structures partition the complete set of states into

subsets, and each set has one representative ("root") of the set. New partitions can be created with some state as a root, the root of any state can be found, we can check if two states are in the same set and partitions can be merged. Several variants of this union-find structure have been proposed and properties have been explored by [Tar75, TvL84] and notably [Blo19] (for concurrent applications).

Name Parallel Depth-First Search
Type Concurrent Depth-First Search for Accepting Cycles
Description This is a simple algorithm using a load-balancer that distributes the work between several threads. An example of this algorithm is shown in [BBDL$^+$18] (Algorithm 12.5). The algorithm itself is trivial, but efficiently implementing (both in terms of time and memory) the load balancing is not and sophisticated techniques [BL99, LvW11, LvW10] are needed to achieve practical speed-up. Improvements have been made by making the DFS order more precise by [APPB10] and [Hol12].

Name Swarmed Nested Depth-First Search [EPY11]
Type Concurrent Swarming Depth-First Search for Accepting Cycles
Description This is a simple variation on the regular NDFS algorithm, but it spawns multiple instances of the search and relies on the non-determinism of the DFS search to increase the chance of an accepting cycle being found. Theoretically, this does not have to lead to any significant speed-up, but in practice it generally does – at the cost of expending more resources to the search (in terms of workers and memory).

Name Parallel Nested Depth-First Search (PNDFS) [ELPvdP12, LLvdP$^+$11]
Type Concurrent Depth-First Search for Accepting Cycles
Description The CNDFS/PNDFS algorithm is a version of NFDS for multi-core architectures partially based on ENDFS [EPY11]. The general idea of these algorithms is to spawn multiple searches (all starting from the initial state) and rely on the non-deterministic nature of the search (which child will be searched next is random) to have one worker prune the search space of another, in many cases speeding up the detection of accepting cycles. This is done by storing the blue colouring locally for each worker, but the red colouring global. A version of CNDFS can be found in [BBDL$^+$18] (Algorithm 12.7), and a version of PNDFS can be found in [Oor19] (Figure 3.3).

Name Fully synchronised SCC-based [Low15]
Type Concurrent Depth-First Search for Strongly Connected Components
Description In this algorithm the search instances do not overlap, so each state is visited by at most one worker. If a worker encounters an unexplored state, it spawns new searches for each of its unexplored children and marks the state as partially explored, or explored if all child searches finished. If a worker encounters a partially explored state, it waits until the state is fully explored before continuing its own search. A side effect of this is that the algorithm may deadlock when all searches are waiting for each other. There are ways to mitigate this problem, but they are expensive in terms of overhead and memory usage, Additionally, this algorithm performs quadratically in a worst-case scenario, instead of linear like Tarjan's algorithm. This algorithm can in certain circumstances create a speedup however, and it generally works best on graphs with few large SCCs.

Name Swarmed SCC-based [RDLKP16]
Type Concurrent Swarming Depth-First Search for Strongly Connected Components
Description Similar to the other SCC-based algorithm and CNDFS, this algorithm spawn multiple instances of the sequential algorithm simultaneously. These instances communicate the

complete SCCs and so hopefully prune the search space of the other searches. These explored SCCs are communicated via a shared data structure [RDLKP16]. There are several optimisations, like sharing *partial* SCCs (see UFSCC), as well as using a lock free union-find structure for the sequential algorithms so that the data can be shared among all workers. This allows other workers to access the found SCCs by others. The algorithm can be found in [BBDL$^+$18] (Algorithm 12.9).

Name union-find SCC-based (UFSCC) [Bv16, BLv16, Blo19, BBDL$^+$18]
Type Concurrent Depth-First Search for Strongly Connected Components
Description This algorithm is based on the concept of swarmed depth-first search, with state space pruning by other workers trough communication of explored parts of the graph. Prior to this algorithm, the latest approach was to only communicate fully explored SCCs with other workers [RDLKP15], while relying on the randomness of the DFS to distribute the workload easily. The problem with that algorithm is that it does not scale for graphs with few large SCCs. UFSCC communicates partial SCCs as soon as they are found, and so scales on even those graphs. Two pseudocode examples of this algorithm can be found in [BBDL$^+$18] (Algorithm 12.10) and [Blo19] (Algorithm 3). While functionally almost equivalent, the notation and naming in both differs significantly (as is the case with the Set-Based SCC algorithm as well).

B LEMMA PROOFS

This appendix provides the proofs for the five lemmas in chapter 4. The notation for the proofs differs from the PVL code. This is done for brevity and readability, as well as for consistency with the previously used notations (for example in chapter 2). Firstly, `rep(N, S, v)` is written as $S(v)$ (note the subtle difference with earlier use of $S(v)$, where it would represent `part(N, S, v)` instead). A path from state v to w is denoted $v \rightarrow^* w$. The elements of an explicit sequence are contained within angled brackets ($\langle ... \rangle$) instead of PVL's square brackets (`[...]`). Indexing a sequence S at index i (in PVL this would be `S[i]`) is written as S_i, and a sub-sequence from j to k (k exclusive, in PVL `S[j..k]`) is $(S)_j^k$. The sequence S with index m updated to v is denoted $S_{m \rightarrow x}$, instead of `S[m -> x]` in PVL. Concatenation of two sequences is done via the $\|$ operator, whereas the PVL syntax would use the + operator. Finally, mathematical first-order logic notation is used, for example $\forall$ instead of `\forall`.

Proofs

Let S be a union-find sequence, and v a state in S. We prove by induction that there exists a path $v \rightarrow^* S(v)$ in S with at least a length of 2.

Base case ($S_v = v$). In this case, $S(v) = v$, and so $v \rightarrow^* S(v) = \langle v, v \rangle$.
Induction hypothesis IH. There exists a path $P = S_v \rightarrow^* S(v)$ in S of at least length 2.
Induction step. For some v, if by IH $P = S_v \rightarrow^* S(v)$ exists, then $\langle v \rangle \| P = v \rightarrow^* S(v)$, so there exists a path $v \rightarrow^* S(v)$ in S. $\square$

Lemma 1. Given a union-find sequence S and a state v, then there exists a path $v \rightarrow^* S(v)$ in S with at least length 2.

Let S be a union-find sequence, and v and r two states in S. We prove by contradiction that if $S_r = r$ and there exists a path $v \rightarrow^* r$ in S with at least a length of 2, then $S(v) = r$.

Contradiction. Assume $S(v) \neq r$. We show using a loop invariant that this implies $S(r) \neq r$, and that this assumption causes a contradiction.

If there exists a path $P = v \rightarrow^* r$ in S with $|P| \geq 2$, then there also exists a path $Q = S_v \rightarrow^* r$ in S, with S_v the successor of v and $|Q| = |P| - 1$. If $|Q| = 1$, then $r = r$. We use this to construct the following loop, where w is initialised to v:

$$\textbf{while } w \neq r \textbf{ do } w = S_w$$

The loop invariant used is $S(w) \neq r$.

Initialisation. The loop invariant is guaranteed before the first iteration of the loop by our assumption that $S(v) \neq r$.
Maintenance. The loop invariant is maintained throughout all iterations of the loop, since by definition $S(S_w) = S(w) \neq r$.
Termination. Because there exists a path $w \to^* r$, each state has only one successor, and there are no cycles in S (see section 4.4.3), eventually $w = r$ and the loop will terminate. At this point, we have maintained the loop invariant and $S(r) \neq r$. This means that if we assume that $S(v) \neq r$ we can prove that $S(r) \neq r$.

Because we defined $S_r = r$, and by definition $S_r = r \iff S(r) = r$, we have proven $S(v) = r$. □

Lemma 2. Given a union-find sequence S and states v and r, where $S_r = r$, and there exists a path $v \to^* r$ in S with at least length 2, then $S(v) = r$.

Let S be a union-find sequence, and $P = v \to^* S(v)$, $|P| \geq 2$ a path in S. We prove by induction on $|P|$ that $\forall i \in P : S(i) = S(v)$.

Base case ($|P| = 2$). The only two elements in P are $P[0] = v$ and $P[1] = S(v)$. By definition, the representatives of v and $S(v)$ are both $S(v)$.
Induction hypothesis IH. $\forall i \in \mathit{tail}(P) : S(i) = S(S_v)$.
Inductive step. For some P with $|P| > 2$, $P = \langle \mathit{head}(P) \rangle \parallel \mathit{tail}(P)$ and $\mathit{head}(P) = v$. Since by definition the representative of both v and S_v is $S(v)$, and by the IH, $\forall i \in P : S(i) = S(v)$. □

Lemma 3. Given a union-find sequence S, a state v and a path $P = v \to^* S(v)$ in S with $|P| >= 2$, then $\forall i \in P : S(i) = S(v)$.

Let S be a union-find sequence, v and w two states in S, and $T = S_{S(w) \to S(v)}$. We prove by contradiction that $\forall i \in [0, N) : S(i) = S(w) \implies T(i) = S(v)$.

We prove that $S(i) = S(w) \implies T(i) = S(v)$ for each $i \in [0, N)$ separately:

Contradiction. Assume for some state $x \in [0, N)$ where $S(x) = S(w)$ that $T(x) \neq S(v)$. We show using lemma 1, lemma 2, and loop invariants that this assumption causes a contradiction.

We first use lemma 1 to show that there exists a path $P = x \to^* S(x)$ in S with at least size 2. Since $S(x) = S(w)$ we know that $P = x \to^* S(w)$ as well. We then use P to construct the following loop with y initialised to 0:

$$\textbf{while } P_y \neq S(w) \textbf{ do } y := y + 1$$

Two loop invariants are used. The first invariant used is $\forall z \in (P)_0^y : S_z = T_z$ **(a)**, and the second is $(P)_0^{y+1} = x \to^* P_y$ in T **(b)**.

Initialisation. With y initialised to 0, loop invariant **(a)** is guaranteed before the first iteration of the loop, since $(P)_0^0 = \emptyset$. Invariant **(b)** is guaranteed since $(P)_0^1 = \langle x \rangle$.
Maintenance. Loop invariant **(a)** is maintained because we know that y is only incremented when $P_y \neq S(w)$. We also know that $T = S_{S(w) \to S(v)}$, and so $P_y \neq S(w) \implies S_{P_y} = T_{P_y}$. Invariant **(b)** is maintained because after the loop body we know by invariant **(a)** that $\forall z \in (P)_0^y : S_z = T_z$. By lemma 1 we know that $P = x \to^* S(x)$ in S, which logically implies $(P)_0^{y+1} = x \to^* P_y$ in S. This fact combined with loop invariant **(a)** maintains loop invariant **(b)**.
Termination. Since $P = x \to^* S(x)$ in S, eventually $y = S(x) = S(w)$ and the loop will

terminate. At this point the loop invariants prove that $(P)_0^{y+1} = x \to^* S(w)$ in T, and because $T_{S(w)} = S(v)$ we can say that $(P)_0^{y+1} \parallel \langle S(v) \rangle = x \to^* S(v)$ in T.

We use this last fact to satisfy the preconditions of lemma 2, to show that $T(x) = S(v)$. We initially assumed $T(x) \neq S(v)$, so we found a contradiction and can conclude that $T(x) = S(v)$ must be true.

As stated before, if we do these steps for each state $i \in [0, N)$ where $S(i) = S(w)$, we have proven that $\forall i \in [0, N) : S(i) = S(w) \implies T(i) = S(v)$. □

Lemma 4. Given a union-find sequence S, two states v and w, and the sequence $T = S_{S(w) \to S(v)}$, then $\forall i \in [0, N) : S(i) = S(w) \implies T(i) = S(v)$.

Let S be a union-find sequence, v and w two states in S, and $T = S_{S(w) \to S(v)}$. We prove by contradiction that $\forall i \in [0, N) : S(i) \neq S(w) \implies T(i) = S(i)$. This proof is similar to that of lemma 4.

We prove that $S(i) \neq S(w) \implies T(i) = S(i)$ for each $i \in [0, N)$ separately:

Contradiction. Assume for some state $x \in [0, N)$ where $S(x) \neq S(w)$ that $T(x) \neq S(x)$. We show using lemma 1, lemma 2, lemma 3 and loop invariants that this assumption causes a contradiction.

We first use lemma 1 to show that there exists a path $P = x \to^* S(x)$ in S with at least size 2. Additionally, by lemma 3 $\forall i \in P : S(i) = S(x)$. We then use P to construct the following loop with y initialised to 0:

$$\textbf{while } P_y \neq S(x) \textbf{ do } y := y + 1$$

Two loop invariants are used. The first invariant used is $\forall z \in (P)_0^y : S_z = T_z$ **(a)**, and the second is $(P)_0^{y+1} = x \to^* P_y$ in T **(b)**.

Initialisation. With y initialised to 0, loop invariant **(a)** is guaranteed before the first iteration of the loop, since $(P)_0^0 = \emptyset$. Invariant **(b)** is guaranteed since $(P)_0^1 = \langle x \rangle$.
Maintenance. Loop invariant **(a)** is maintained because we know by lemma 3 that $\forall i \in P : S(i) = S(x)$ and our assumption states $S(x) \neq S(w)$, so $i \neq S(w)$ since $S(S(w)) = S(w)$. We also know that $T = S_{S(w) \to S(v)}$, and so $P_y \neq S(w) \implies S_{P_y} = T_{P_y}$. Invariant **(b)** is maintained because after the loop body we know by invariant **(a)** that $\forall z \in (P)_0^y : S_z = T_z$. By lemma 1 we know that $P = x \to^* S(x)$ in S, which logically implies $(P)_0^{y+1} = x \to^* P_y$ in S. This fact combined with loop invariant **(a)** maintains loop invariant **(b)**.
Termination. Since $P = x \to^* S(x)$ in S, eventually $y = S(x)$ and the loop will terminate. At this point the loop invariants prove that $(P)_0^{y+1} = x \to^* S(x)$ in T.

We use this last fact to satisfy the preconditions of lemma 2, to show that $T(x) = S(x)$. We initially assumed $T(x) \neq S(x)$, so we found a contradiction and can conclude that $T(x) = S(x)$ must be true.

As stated before, if we do these steps for each state $i \in [0, N)$ where $S(i) = S(w)$, we have proven that $\forall i \in [0, N) : S(i) \neq S(w) \implies T(i) = S(i)$. □

Lemma 5. Given a union-find sequence S, two states v and w, and the sequence $T = S_{S(w) \to S(v)}$, then $\forall i \in [0, N) : S(i) \neq S(w) \implies T(i) = S(i)$.

C SIMPLE REFINEMENT OF A UNION-FIND DATA STRUCTURE

Lammich [Lam19] describes a way of using Isabelle/HOL and the Isabelle Refinement Framework to verify imperative programs using refinement techniques. As an example the refinement of a union-find data structure from an equivalence relation to an array-based implementation (similar to the implementation in 4.4.3) is given. This example can be done using a “simple” approach. This approach is limited, but its relative simplicity means that implementing something similar in VerCors is feasible. The limitations of this approach are mitigated by using the Isabelle Refinement Framework. Below this simple approach is explained using the union-find example from [Lam19], translated to “pseudocode” PVL (i.e. *not verified or tested*).

Lammich defines 4 HOL functions: $find_1$, $invar_1$, α_1 and $union_1$. Here, $invar_1$ is an invariant over the union-find, $alpha_1$ is a mapping from a sequence to an equivalence relation, and $find_1$ and $union_1$ correspond to the abstract functions $find$ and *union*. In listing 20 these are translated to PVL as pure and abstract functions, respectively. Note that all of the implemented functions are very similar to the functions and methods in 4.4.3. An important difference between this code and the HOL example is that we skip an extra refinement step in [Lam19], where a union-find as a *list* is refined to an implementation using an *array*. Since there are no lists or arrays in PVL, only sequences, this step is not needed.

The abstract methods have a method contract that defines the correct behaviour of the operations. Since the methods are abstract, this correctness can be assumed and does not have to be proven. If we then want to verify the correctness of the implemented `union_1` function, we can simply use the definition of *union* in a postcondition to `union_1` to accomplish this, as shown in listing 21. It is possible that there is additional work needed to prove the postconditions of `union_1`, but the correctness criteria of the operation do not have to be formulated again since we defined `alpha_1` as the mapping from a sequence to an equivalence relation. This same method can also be applied to the `find_1` function.

Listing 20: *Union find from [Lam19] in VerCors*

```
// the abstract functions, R is a set<set<int>>

// ... the find method contract, defines correctness
int find(set<set<int>> R, int i);
// ... the union method contract, defines correctness
set<set<int>> union(set<set<int>> R, int x, int y);

// the implementation, l is a seq<int>

pure int find_1(seq<int> l, int i) = l[i] == i ? i : find_1(l, l[i]);
pure boolean invar_1(seq<int> l) = (\forall int i; i < |l|; l[i] < |l|);
pure void alpha_1(seq<int> l) = {{x, y} | x < |l| && y < |l| && find_1(l,
    x) == find_1(l, y)};
pure seq<int> union_1(seq<int> l, int x, int y) = l[find_1(l, x) ->
    find_1(l, y)];
```

Listing 21: *Verification of union_1 using* `union`

```
requires invar_1(l);
requires 0 <= x && x < |l|;
requires 0 <= y && y < |l|;
ensures invar_1(\result);
ensures alpha_1(\result) == union(alpha_1(l), x, y);
pure seq<int> union_1(seq<int> l, int x, int y) = l[find_1(l, x) ->
    find_1(l, y)];
```