Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Dependency Analysis in Automated Verification Proofs

Master's Thesis

Andrea Keusch

Wednesday 1st October, 2025

Advisors: João Pereira, Lea Salome Brugger, Prof. Dr. Peter Müller

Programming Methodology Group, ETH Zürich

In the writing process of this thesis, ChatGPT was used as a thesaurus and translator.

Acknowledgments

I would like to thank all people who supported me throughout this thesis. First and foremost, to João Pereira and Lea Brugger who made time for weekly meetings providing helpful advise and valuable feedback. I was always looking forward to our meetings and left them feeling motivated to tackle the next step. I would also like to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this project. Lastly, I am deeply grateful to my friends and family for their lasting support not only during this thesis but also over many years before that. Thank you for always believing in me.

Abstract

Automated deductive verification proofs are complex, thus, making it hard to understand and explain why a program does or does not verify against its specification. In particular, the facts known to the verifier at a given point are not represented explicitly which hinders reconstruction of the intent that led to a program being annotated a certain way. As a result, it is hard to fix proofs when they break due to changes in tooling and to evolve them by proving additional properties. Further, debugging failures is cumbersome and tracking verification progress, i.e., estimating the completeness of the specification, is challenging. To overcome these challenges, we propose a dependency analysis algorithm that automatically detects for each proof obligation which assumptions, explicit or implicit, were used to prove it. It reveals direct and indirect dependencies within and across program components, thus breaking modularity. Dependencies are represented as a user-level dependency graph where nodes correspond to source code fragments and edges indicate their direct dependencies.

We implement and integrate this dependency analysis into Viper's symbolic execution backend and provide tool support to query and visualize dependency graphs. The usefulness of our approach is dependent on how precise its results are: ideally, our algorithm should report only true dependencies. However, this is not always possible. We identified multiple causes of imprecision and already found solutions to some of them. Performance is reasonable for medium-sized programs but currently scales poorly to large-scale verification projects.

The dependency graph, once computed, can be queried efficiently and reveals insightful information for various use cases such as impact estimation of invalid assumptions, detection of irrelevant, uncovered assumptions, and verification progress tracking in terms of proof coverage.

Contents

Chapter 1

Introduction

The common strategy to gain confidence about the correctness of a program is testing against some exemplary inputs. However, tests cannot prove the absence of bugs unless every single possible input is tested, which is not feasible in general. Stronger guarantees can be achieved by formally proving the correctness of a program with respect to a formal specification. In automated deductive verification, such as Viper [28], specifications are given as proof annotations which are then used to automatically verify the program. Unfortunately, verification proofs are often complex and hard to understand; for example, pointing out the assumptions required to prove an obligation is not trivial. Detecting such dependencies between assumptions and assertions has several practical applications, some of which are presented in the following:

1. **Proof Understanding and Explainability.** Identifying and visualizing dependencies between assumptions and proof obligations provides essential information, such as a justification of why the obligation verifies, which facilitates understanding and explainability of verification proofs. This is particularly important in real-world verification projects which have to be maintained over many years, usually by dynamic project teams. For members that are new to the project or extend other people's work, it is essential to have efficient techniques to gain a deep understanding of existing proofs. Further, deeper understanding and visualizations of dependencies facilitates collaboration between verification engineers and non-experts in verification and enables broader adoption of verification by making it more accessible.
2. **Assumption Impact Estimation.** In practice, verified programs often have some unverified components introducing unverified assumptions in the form of postconditions. These assumptions might turn out to be invalid; for example, a postcondition of an unverified method might not hold due to a bug in the code. It is crucial to estimate the impact

of such invalid assumptions by analyzing which program components and obligations are affected. This requires non-modular reasoning. The common procedure is to remove the invalid assumption and re-run verification. By revealing dependencies in verification proofs, the impact of multiple assumptions can be analyzed in a single verification run, namely by finding the dependents of each assumption.

3. **Debugging and Proof Repair.** Verification errors can arise when updating the specification of a program, e.g., by adding proof obligations such as postconditions, or due to proof instability [34, 35]. In the latter case, failures occur due to modifications to the code, its dependencies, the verification tool, or its configuration, even if the changes are minor and should not affect the verification outcome. Debugging such failures is challenging.
   Information about dependencies can be used to support the debugging process in various ways: First, it allows users to gain useful insights into the previously successful verification; for example, by understanding the dependencies required to verify a proof obligation. Second, dependencies in partially verified programs might reveal helpful information. While the question of why verification failed cannot be answered directly, dependencies still provide insights about the verified obligations up to the point of verification failure. Lastly, the ability to detect redundant or unnecessary assumptions can be exploited for debugging purposes: Explicit assumptions can be added such that the program verifies. Dependencies discovered for this new program reveal which of the assumptions are required by the proof. Consequently, the user can remove the unnecessary assumptions and, more importantly, learns what is missing for successful verification, namely the remaining assumptions. Manually manipulating a program that fails to verify by iteratively adding and removing assumptions, re-running verification after each modification, is already common practice when debugging verification failures. However, while feasible, it is inefficient and cumbersome.

4. **Removing and Weakening Assumptions.** Information about unnecessary, redundant assumptions can be used, for example, to clean up the code and weaken the assumptions, thus strengthening the proof. Weakening preconditions additionally removes some burden from the caller side as fewer constraints need to be ensured when calling the method.

5. **Verification Progress.** Verification projects advance by progressively specifying and verifying stronger properties. In large-scale verification projects, tracking progress is crucial for project management but challenging because manual analysis is not feasible and existing au-

tomated solutions have significant shortcomings. In software testing, the progress of creating a thorough test suite is assessed by test coverage metrics. Analogously, a notion of proof coverage could assist in quantifying verification progress. By detecting all dependencies, we reveal assumptions and statements that are not required by any proof obligation, i.e., they are uncovered. Computing proof coverage is therefore a direct application of dependency discovery.

In summary, various aspects of the verification process could be enhanced by identifying dependencies between assumptions and proof obligations within and across program components. Doing such an analysis manually requires considerable effort and is error-prone. First, one has to account for all possible paths leading up to the obligation at hand. Second, it can be difficult to keep an overview over the numerous assumptions introduced throughout verification: On the one hand, proof annotations *explicitly* state assumptions and proof obligations in the form of pre- and postconditions as well as `assume` statements. In addition to that, most statements modify the program state which can be seen as adding *implicit* assumptions; for example, the equality resulting from an assignment. Third, finding dependencies across functions, methods, and modules requires non-modular reasoning, which increases the search space tremendously.

Our main goal is to develop an automated analysis to discover such dependencies. This analysis should support the previously mentioned use cases, meaning that it should be applicable to partially verified programs and provide a proof coverage metric. The analysis should be sound, efficient, and as precise as possible. Furthermore, users should be provided with an interface for querying the analysis results.

Contributions.
Our contributions can be summarized as follows:

1. We provide an intuitive definition of semantic dependencies within and across program components in verification proofs.
2. We design and implement a sound and precise algorithm to automatically detect these dependencies in Viper's symbolic execution backend.
3. We formally define a proof coverage metric for methods and individual assertions, which can be computed based on the dependency analysis results.
4. We provide a user interface to analyze the dependencies and extract useful metrics, such as proof coverage, with minimal manual effort.
5. We conduct a thorough evaluation of various aspects of our analysis, namely soundness, precision, and performance.

Chapter 2

Background

This section provides the relevant background for this thesis bu we assume the reader is already familiar with the basics of symbolic execution [9,26] and SMT solvers [18]. We briefly cover Viper, a verification infrastructure, and Silicon, one of Viper's backends. We also address Silicon's heap model. Further, we give a short introduction into SMT solvers and UNSAT cores, which is the foundation of how we detect dependencies. Lastly, we address how Viper is used to verify real-world programs through frontends.

2.1 Viper

Viper [28] is a verification infrastructure that implements two backends for verifying programs with respect to their specification: the symbolic execution backend (Silicon) and the verification condition generation backend (Carbon). For this thesis, we only consider Silicon which is described in the subsequent sections. Viper is also an intermediate verification language with object-oriented and imperative features. Viper programs consist of collections of methods and functions containing verification annotations which make up their specification. More precisely, each method and function is specified by a contract with *preconditions*, which need to be ensured on every call, and *postconditions*, which are guaranteed to hold at termination. A successfully verified Viper program guarantees that all concrete executions satisfy the specification. Concrete executions are defined by sequences of operations given in the form of statements and expressions. *Expressions* are side-effect free and can be evaluated to a value, while *statements* modify the state and are composed of expressions.

The Structure of a Viper Program

A Viper program consists of a collection of program components, namely, methods, functions, and domains. They interact with each other over their

specification. More concretely, each component exposes its specification in the form of pre- and postconditions to other components. Preconditions are declared by using the `requires` clause and postconditions by the `ensures` clause. When calling a function or method, the specification can be assumed without verifying it. By providing specifications but hiding implementation details, each component can be verified separately, i.e., modular verification is enabled. Components which access the heap are referred to as "impure" components, otherwise they are "pure".

Methods are defined by pre- and postconditions as well as the body which consists of a sequence of statements. The body can have side-effects, e.g., by modifying the heap, and it can return values. A method verifies successfully if, given that the preconditions hold, the body verifies and guarantees all postconditions on termination. If a method does not have a body, it verifies trivially as long as it is well-formed. We refer to this as an unverified, trusted method.

Functions are defined by pre- and postconditions as well as the body which consists of one single expression. Thus, a function cannot have side-effects but it can be recursive. Verification of a function succeeds if the preconditions and body together imply the postconditions. Unverified, trusted functions do not have a body and verify trivially as long as they are well-formed.

Domains provide the option to encode additional types, for example, arrays or strings. They consist of a set of domain function declarations whose specification are defined by domain axioms. Importantly, domain functions cannot have bodies or pre- or postconditions, thus they cannot access the heap. Domain axioms are boolean expressions which define the behavior of domain functions. Other components use these axioms to reason about calls made to the domain functions.

Reasoning about loops requires the definition of expressions that are guaranteed to hold before and after every iteration, so-called *loop invariants*. Simply speaking, verification of a loop requires that all loop invariants hold before the loop and are preserved by the loop body. Invariants can then be assumed to hold after the loop terminates.

Permission Model

Viper programs can access and modify the heap. This imposes several challenges since references to the heap can be aliases of each other and, thus, point to the same heap locations. To reason about such cases and guarantee memory safety, Viper introduces an expressive permission model which is based on separation logic and fractional permissions.

A heap location can be modified if and only if *full* permission, corresponding to 1, to this location is held. On the other hand, reading a heap location

is allowed when at least *some* permission is held. It is not possible to hold negative or more than full permission to a heap location. Each statement and expression must ensure that its permission requirements are satisfied, otherwise verification fails. This also includes well-formedness constraints enforced on some assertions; for example, a postcondition can only make constraints on a heap reference if it guarantees access to the corresponding heap location. Gaining and losing permission to a heap location is referred to as *inhaling* and *exhaling* permission, respectively. Permissions can be transferred to other components through preconditions of method calls and are returned via postconditions.

Fractional permissions enable components to transfer read permission to other components while keeping part of the permission to that heap location. This guarantees that the value stored in the location cannot be changed. More generally, if full permission to a heap location is held, it is guaranteed that no other components holds *any* permission to this location, ensuring the absence of race conditions. Viper encodes this using the frame rule of separation logic which states that a heap location's value is preserved as long as some permission to that location is held.

In Viper, heap references are denoted by `x.f`, where `x` is a reference and `f` is a field. The access predicate `acc(x.f, p)` denotes `p` permission to heap resource `x.f` where `p` is a fractional permission amount, e.g., `1/2`. Omitting `p` in the access predicate defaults to full permission. We use `Z`, `W` to indicate zero and full permission, respectively. Further, `wildcard` denotes any permission greater than `Z` and users can declare permission variables of type `Perm`. The expression `perm(x.f)` returns the currently held permission to `x.f`. Viper provides explicit `inhale` and `exhale` statements, e.g., `inhale acc(x.f, 1/2)` and `exhale acc(x.f)`, but permissions are also gained or lost through pre- and postconditions, method calls, predicate folds, and more. The `inhale` statement is equivalent to the `assume` statement but additionally adds permissions as declared. The `exhale` statement verifies that the exhaled expression holds and removes the declared permissions. Lastly, the `quasihavoc` statement invalidates the values of the specified heap locations such that no constraints on their values are known after the havoc operation.

To enable reasoning about unbounded data structures, Viper introduces quantified permissions, predicates, and magic wands. *Quantified permissions* can be used to model access to multiple heap locations at once. Details are not relevant for this thesis. A *predicate* is a potentially recursive, parameterized assertion as depicted in Listing 2.1. It is encoded as a heap resource, thus, (partial) access to a predicate can be inhaled and exhaled. Predicate instances can be exchanged with their bodies using the `unfold` statement. The opposite direction is called fold. Further, the `unfolding P in E` expression temporarily unfolds predicate instance `P` in order to evaluate `E` but does not exhale it.

Lastly, *magic wands* provide the means to package permissions to a partial data structure and unpackage them later on to get back permission to the full data structure.

```
1  field next: Ref
2
3  predicate list(this: Ref) {
4    acc(this.next) &&
5    (this.next != null ==> list(this.next))
6  }
7
8  method clearList(ls: Ref)
9      requires list(ls)
10     ensures  list(ls)
11 {
12     unfold list(ls)
13     ls.next := null
14     fold list(ls)
15 }
```

Listing 2.1: A Viper predicate encoding an unbounded list. In `clearList`, unfolding the list predicate inhales access to the first list element and potentially to the list's tail. The subsequent `fold` statement exhales permission to the first element.

2.2 Viper's Symbolic Execution Backend

As already mentioned, Viper has two backends, one of which is discussed in this section. The symbolic execution backend [31], called Silicon, takes a Viper program and verifies it through symbolic execution, as the name suggests. It operates on a *symbolic state* that captures all constraints on program variables and heap locations learned during symbolic execution.

The symbolic state consists of three components. First, the *store* keeps track of the mapping from program variables to their (symbolic) value which can be a concrete value, an *internal variable*, or an *internal term*. In contrast to Viper variables, internal variables are immutable and, thus, keep track of different versions of Viper variables. Fresh, internal variables are created, for example, when a Viper variable is modified due to an assignment. Internal terms are expressions over internal variables. The store is updated whenever a program variable gets assigned to. Second, the *path condition* maintains the collection of all constraints learned so far on the current path. This includes branch and loop conditions as well as equalities learned from assignments, postconditions from method calls, and so forth. Note that these constraints are added in the form of internal terms. The third component is the symbolic heap which we cover in Section 2.3 (Symbolic Heap Model).

Due to the modular verification strategy, Silicon initializes one verifier instance for each program component. The verifier symbolically executes each path of the given program component and reports the verification result. To symbolically execute a statement, Silicon first evaluates all its expressions to internal terms. Next, Silicon verifies that all requirements enforced by the statement itself or any of its expression are met; for example, that full permission to the assigned heap location is held. In many cases, an SMT solver is invoked in order to prove the requirements. We will go into more details in Section 2.4 (SMT Solver and UNSAT Cores). If some requirements cannot be proven, verification fails. Otherwise, the symbolic state is updated according to the statement's semantics, for example, by adding assumptions to the path condition or updating the store.

We abstain from formally defining the requirements and semantics of each statement but give a quick overview of the relevant details. Expressions need to satisfy their permission requirements and guarantee that all operations are executed within their domain, i.e., the preconditions of all operations and function calls need to be satisfied. Statements always require all of its expressions to verify. Declarations, `assume` and `inhale` statements have no further requirements. They add the corresponding assumption to the path condition and potentially chunks to the symbolic heap. `Assert` statements check whether their expressions are guaranteed to hold but they do not modify the symbolic state. Assignments require write access to the left-hand side and modify the symbolic state by updating the left-hand side's value.

Method calls need to ensure all preconditions and exhale them, thus, potentially transferring permission. Then, they inhale the postconditions, potentially receiving permission, and they assign the return value to the left-hand side (if there is one).

`Unfold` statements exhale the predicate instance and inhale the predicate body instead. The counterpart, the `fold` statement, exhales the predicate body, thus, verifying that the body is satisfied, and then inhales a predicate instance. Note that we omit a few cases that are not relevant for this thesis, in particular, `apply` and `package` statements.

`Quasihavoc` statements require write access to the specified heap locations and invalidate their values, i.e., the heap locations are assigned fresh, internal variables for which no constraints on the values are known. This is implemented through exhaling and inhaling full permission to the heap locations.

Symbolic execution deals with branches by forking the execution. One verifier instance assumes the branch condition and executes the if-branch. The second verifier instance assumes the negation of the branch condition and executes the else-branch. In its default configuration, Silicon never joins paths, i.e., each verifier instance steps through the remainder of the program separately, potentially branching repeatedly. Silicon invokes feasibility checks

before branching and only executes feasible paths, i.e., branches where no contradiction has been found. For infeasible paths, verification is terminated immediately since they verify trivially. This performance optimization is a measure to avoid path explosion.

Loops are internally encoded using branches. Intuitively speaking, one branch ensures that the loop invariants hold before the loop, that all loop invariants are preserved by the loop body, and then terminates. The second branch sets up the symbolic state after the loop which includes invalidating the values of variables and heap locations modified by the loop, assuming the loop invariants as well as the negation of the loop condition. This branch continues verification of the program after the loop. The complete encoding is more involved but details are not relevant for this thesis.

2.3 Symbolic Heap Model

We already addressed Viper's permission model. In this section, we explain how this permission model is integrated into Silicon's symbolic execution algorithm.

The *symbolic heap* keeps track of all available heap resources. It is a map from Viper heap references, e.g., `x.f`, to so-called heap *chunks*. Chunks are immutable objects capturing the value and permission amount to a heap location. In this thesis, we use the notation `Chunk(v, p)` to denote a chunk storing value `v` and with permission amount `p`. Multiple heap references can point to the same chunk when they are aliases.

Viper defines different kinds of heap locations, namely fields, predicates, and magic wands, all of which can be non-quantified or quantified. They are captured by different types of chunks. Since all kinds of chunks offer the same operations, we do not distinguish between these kinds in this thesis.

To verify permission requirements imposed by statements and expressions, Silicon tries to find a suitable chunk. That is, a chunk that corresponds to the given heap location and provides at least the required permission amount. If such a chunk cannot be found, verification fails. When inhaling and exhaling permission to a heap location, a suitable chunk is added and removed, respectively. Since chunks are immutable, updates to the heap, e.g., assigning a new value or exhaling partial permission, are encoded by replacing the existing chunk by a new chunk.

The heap imposes several challenges in verification tools. In particular, heap references can be aliases of each other. This fact might not be known right from the beginning and can be learned at any point throughout verification; for example, when entering a branch with condition $x = y$. To capture this, Silicon merges chunks once aliasing can be proven. Further, it explicitly adds

the assumption $x \neq y$ when non-aliasing is detected, which happens, for example, when their permissions would add up to more than 1. To make verification more complete, internal operations like *state consolidation* and *heap summary* execute such checks and modify the symbolic state accordingly. These operations can be triggered at any point during verification and are hidden from users.

2.4 SMT Solver and UNSAT Cores

SMT solvers, such as Z3 [18], determine whether a first-order formula is satisfiable. Since this problem is undecidable for many theories, SMT solvers are incomplete, i.e., they might not come to a conclusion and return "unknown" instead. Many deductive verifiers rely on SMT solvers to automatically check whether the proof obligations hold. Silicon is one such tool that outsources the verification of most assertions to an SMT solver.

To this end, everything added to the path condition is also sent to the SMT solver. That is, the SMT solver learns all constraints on internal variables introduced throughout verification. As a consequence, the SMT solver can be queried to verify proof obligations on these internal variables.

To verify a proof obligation, its negation is sent to the SMT solver. If unsatisfiability of this negation can be proven, the obligation is valid, i.e., is guaranteed to hold. As proof of unsatisfiability, the SMT solver provides an *UNSAT core* containing all assumptions used to proof unsatisfiability. In terms of the original (non-negated) obligation, the UNSAT core consists of all assumptions required to prove the obligation; for example, the proof of assertion `x > 0` might have the UNSAT core `(x == a, a > 10)`.

2.5 Viper Frontends and Gobra

Viper is a program verification language that cannot be executed concretely. Instead, support of real-world programming languages is provided through Viper frontends. These frontends correspond to programs in the target language and are enhanced with verification annotations, such as preconditions or loop invariants. Verification of a frontend program proceeds by translating it to a Viper program and verifying it using one of the two backends, e.g., Silicon. The Viper frontend for Go is called Gobra [33] and is used, for example, in the VerifiedSCION project [3]. Further details about Gobra and its encoding to Viper are not relevant for this thesis and thus omitted.

Chapter 3

Definitions

In Chapter 1 (Introduction), we motivated why discovering dependencies in program verification proofs is useful in practice. In this chapter, we define what kind of dependencies we are looking for. We take a top-down approach and start with defining the dependency graph. Then, we describe its nodes, which correspond to assumptions and assertions, followed by the edges, i.e., the dependencies. We give intuitive, informal definitions for them. Lastly, we define the soundness and precision property of the desired dependency analysis algorithm that builds a dependency graph of the input program.

The following definitions were developed with Viper and its symbolic execution backend in mind. However, they are applicable to other deductive program verifiers. All given examples are written in Viper.

3.1 Dependency Graph

We define dependencies present in a program P using a graph representation that corresponds to an abstracted form of program P.

Definition 3.1 (Dependency Graph)
The dependency graph (V, E) of a program P consists of

- *a set of vertices V, which correspond to the set of all nodes of program P, and*
- *a set of edges E, representing the direct dependencies between the nodes.*

Contrary to what one might expect, the vertices in V do not always correspond to statements since, for example, we also need to track invariants, which are expressions. Therefore, dependency nodes are defined as follows:

Definition 3.2 (Dependency Node)
A dependency node is any of the following language constructs, where top level conjunctions have been split into individual constraints:

- *primitive statements, i.e., statements that do not contain other statements like branches or loops do,*
- *branch and loop conditions as well as their negation,*
- *invariants,*
- *pre- and postconditions, and*
- *assumptions exposed by a program component to be used by other components (e.g., in Viper these are postconditions, function bodies, and domain axioms).*

Splitting of top-level conjuncts means, for instance, a conjunction $acc(x.f) \wedge x.f > 0$ is represented by two nodes, namely $acc(x.f)$ and $x.f > 0$. This allows us to define dependencies more precisely since we can distinguish the case of having a dependency to both nodes from depending on just the access permission, for example. Branches and loops are excluded because instead we include their conditions and invariants. This approach is equally expressive and has the advantage that all nodes are atomic, i.e., they do not contain subnodes.

3.2 Assumptions and Assertions

The previously described dependency nodes can take the role of assumptions, assertions, or both at the same time. Which of the three cases applies depends on the node and the context, i.e., in what relation the node stands to the analyzed assertion. An example, which is explained in detail throughout this section, is given in Listing 3.1, page 13.

Definition 3.3 (Assumption)
A dependency node can act as an assumption if it adds new facts to the proof context.

For example, an assignment (e.g., Listing 3.1, lines 7, 8, 14, 15) adds a fact stating the equality of the left- and right-hand side and is, therefore, an assumption. Explicit `assume` and `inhale` statements, preconditions (e.g., Listing 3.1, lines 2 and 3), and branch and loop conditions (e.g., Listing 3.1 line 9) are assumptions introducing the corresponding expression as a new fact. Almost every node changes the proof context and can thus act as an assumption. Exceptions are, for example, explicit `assert` statements (e.g., Listing 3.1, line 18). Note that postconditions (e.g., Listing 3.1, lines 4 and 5) are assumptions because they are assumed by callers of the method.

Definition 3.4 (Assertion)
A dependency node can act as an assertion if it enforces some property on the proof context and causes a verification failure if that property cannot be proven.

For example, nodes accessing or modifying a heap resource enforce some permission requirements (e.g., Listing 3.1, line 15). Obviously, explicit asser-

tions and postconditions (e.g., Listing 3.1, lines 5, 18) are assertions which verify the corresponding expression.

```
1     method foo(n: Int, res: Ref)
2         requires n > 0      // (A)
3         requires acc(res.f) // (A)
4         ensures  acc(res.f) // (A), (Q)
5         ensures  res.f > 0 // (A), (Q)
6     {
7         var i: Int := 0 // (A)
8         res.f := n      // (A)
9         while(i < n)    // (A)
10            invariant i <= n     // (A), (Q)
11            invariant acc(res.f) // (A), (Q)
12            invariant res.f > 0  // (A), (Q)
13        {
14            i := i + 1          // (A)
15            res.f := res.f + i // (A), (Q)
16        }
17
18        assert i == n  // (Q)
19    }
```

Listing 3.1: A program with nodes annotated as assumptions (A), assertions (Q), or both, depending on what role they can take according to our definitions.

A node can act as an assumption and an assertion at the same time. For example, the field assignment in Listing 3.1, line 15, enforces some permission requirement and introduces an assumption like any assignment does. Another example are loop invariants (e.g., Listing 3.1, lines 10-12) which need to be asserted and are assumed at different points during verification. As briefly mentioned already, postconditions are assertions and assumptions since we consider dependencies across program components.

3.3 Dependency

Having defined assumptions and assertions, the next step is to state what it means to have a dependency between them. Dependencies are always defined with respect to a program and a verified assertion appearing in that program. We start by defining direct dependencies.

Definition 3.5 (Direct Dependency)
Given a program P with a set of assumption nodes A and a successfully verified assertion Q, $a \in A$ is a direct dependency of Q, denoted $a \rightsquigarrow Q$, iff any of the facts added to the proof context by a is required to semantically prove[1] *any of the*

[1]This excludes assertions needed to trigger quantifier instantiations.

properties enforced by Q on any program path.

A small example is given in Listing 3.2. According to the definition, the assertion directly depends on lines 2 and 3 because their added facts imply the assertion, i.e., $(x.f = 0 \land a = x.f) \Rightarrow a = 0$. Line 1, on the other hand, is not a direct dependency because the assertion does not access any heap resources and hence does not enforce any permission requirements. For our use cases, this is a problem since the assumptions coming from lines 2 and 3 can be introduced only because those nodes' requirements, namely access to $x.f$, were met. Otherwise, verification would fail for those lines. Therefore, the assertion should indirectly depend on line 1.

```
1    var x: Ref := new(f)
2    x.f := 0
3    var a: Int := x.f
4    assert a == 0
```

Listing 3.2: Viper example demonstrating the need of transitivity.

More generally, dependencies of dependencies should be taken into account. We conclude that the dependencies of an assertion should be transitively closed, which leads us to the following definition.

Definition 3.6 (Dependency Set)
Given a program P with a set of assumption nodes A and verified assertion Q, a dependency set of Q is a set of nodes $D \subseteq A$ that includes

(1.) all direct dependencies of Q,

(2.) all direct dependencies of any node $n \in D$, and

(3.) all nodes representing an assumption exposed by a program component (e.g., in Viper, these are postconditions of methods and functions, function bodies, and domain axioms) which is required to prove any assertion in D

*We use notation $D \models Q$ to state that D is a dependency set of Q and notation $D' \not\models Q$ to indicate that D' is **not** a dependency set of Q, i.e., that some obligation coming from Q cannot be proven.*

(2) enforces transitive closure by including indirect dependencies, i.e., dependencies of dependencies, and (3) is required to reason about dependencies across program components, e.g., dependencies to assumptions made in a callee's body.

A trivial dependency set is formed by taking $D = A$, i.e., all assumptions of the original program P. However, our goal is to report as few dependencies as possible, which requires the notion of minimal dependency sets.

Definition 3.7 (Minimal Dependency Set)
A dependency set D is minimal iff

$$\forall D' \subset D.\ D' \not\models Q$$

We use notation $D \models_{min} Q$ *to state that D is a minimal dependency set of Q.*

Note that here, minimal refers to a local minimum. Minimal dependency sets are not always unique, as can be seen in Listing 3.3 where a minimal dependency set for assertion (Q) contains any of the two assumptions but not both of them. We ignore this detail in further discussions since it is not relevant for this thesis. In other words, we consider every minimal dependency set to be equally suitable.

```
1    var a: Int
2    assume a > 0
3    assume a > 10
4    assert a >= 0 // (Q)
```

Listing 3.3: A Viper example with two minimal dependency sets for assertion (Q).

To illustrate a dependency set on a concrete example, consider Listing 3.4, page 16. Here, all nodes belonging to the dependency set of assertion (Q) on line 25 are annotated with (D). Additionally, the dependency set is visualized as a graph in Figure 3.1, page 16, where edges indicate direct dependencies.

Assertion (Q) depends, among other nodes, on one of `inc`'s postconditions, namely line 9. In the graph, we visualized this as a transitive dependency via the method call to `inc`. This postcondition (line 9 in `inc`) directly depends on the assignment, `assume` statement, and on having access to `out.val` which is provided by a precondition. Indirectly, via the assignment and assumption, there is a dependency to the precondition providing access to the heap resource `inp.val`. Since assertion (Q) depends on this postcondition on line 9, it inherits all of these dependencies.

Note that there is no dependency to `inc`'s postcondition on line 8, for example, since it is not required to prove assertion (Q). Moreover, there are some explicit assumptions which are irrelevant, namely the ones not annotated by (D).

```
1  field val: Int
2
3  method inc(inp: Ref, out: Ref)
4    requires acc(inp.val, 1/2) // (D)
5    requires acc(out.val)      // (D)
6    ensures  acc(inp.val, 1/2)
7    ensures  acc(out.val)      // (D)
8    ensures  out.val == inp.val + 1
9    ensures  out.val > 0       // (D)
10 {
11   assume inp.val >= 0    // (D)
12   assume inp.val < 100
13
14   out.val := inp.val + 1 // (D)
15 }
16
17 method client(){
18   var inp: Ref, out: Ref
19   inp := new(val)     // (D)
20   out := new(val)     // (D)
21   assume inp.val < 10
22
23   inc(inp, out)       // (D)
24
25   assert out.val > 0  // (Q)
26 }
```

Listing 3.4: A Viper program with an assertion (Q) and its dependencies (D).

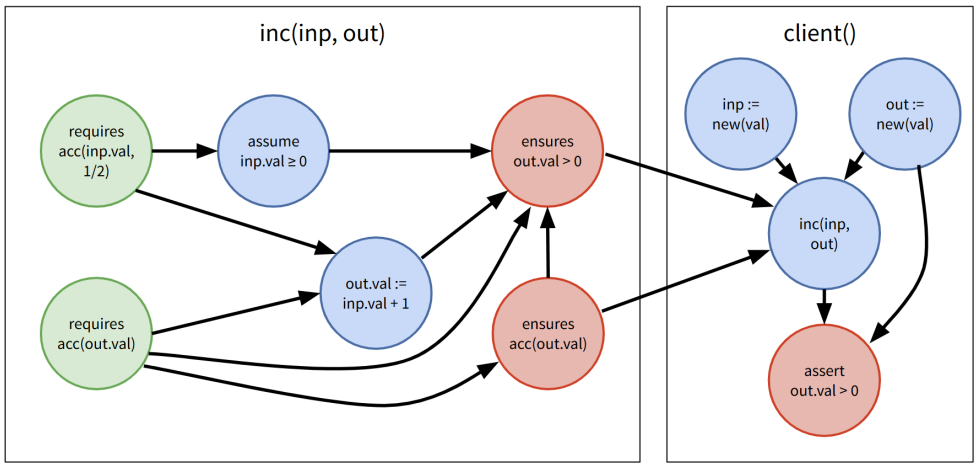

Figure 3.1: The graph representation of the dependency set of assertion (Q) in Listing 3.4.

It is worth noting that there is a direct dependency between two postconditions of `inc`, namely `acc(out.val)` $\rightsquigarrow$ `out.val > 0`, as displayed in Figure 3.1. This dependency results from well-formedness checks, which require post-

conditions to provide access to all resources appearing in the postconditions.

While discussing the example, we already informally introduced the notion of dependency, which we formalize now.

Definition 3.8 (Dependency)
Given a program P with a set of assumption nodes A and a successfully verified assertion Q as well as a minimal dependency set $D \models_{min} Q$,

$$a \in A \text{ is a dependency of } Q, \text{ denoted } a \rightarrow Q,$$
$$\Longleftrightarrow$$
$$a \in D$$

Assuming that a minimal dependency set is intentional. An alternative definition of dependency could be as follows: a is a dependency of Q if there exists a dependency set D where $a \in D$. Referring back to Listing 3.3, page 15, using the alternative definition, both assumptions would be considered to be dependencies of the assertion. With Definition 3.8, on the other hand, only one is a dependency. Which one depends on the chosen minimal dependency set. Since our goal is to report *some* minimal dependency set, Definition 3.8, page 17, is the better choice.

Lastly, we introduce the counterpart of dependencies, namely dependents.

Definition 3.9 (Dependent)
Given program P, assertion Q is a dependent of assumption a, denoted $Q \leftarrow a$, iff a is a dependency of Q.

This concludes the description of the expected result of a dependency analysis. It is representable by a graph with nodes that correspond to assumptions and assertions as well as edges that represent the direct dependencies. Dependency sets correspond to transitive closures.

3.4 Soundness and Precision

So far, we described the expected output of a dependency analysis, namely a graph representing the program and the dependencies between the program nodes. Next, we give formal definitions of the two key properties of a dependency analysis algorithm, namely soundness and precision. They follow directly from Definition 3.6, page 14, and Definition 3.7, page 15.

Definition 3.10 (Soundness)
A dependency analysis algorithm is sound iff for all programs P and all assertions Q, it returns a dependency set of Q, i.e., a set D such that $D \models Q$.

Definition 3.11 (Precision)
A dependency analysis algorithm is precise iff for all programs P and all assertions Q, it returns a minimal dependency set of Q, i.e., a set D such that $D \models_{min} Q$.

3.5 Discussion and Refined Definitions

Dependencies have been defined as everything required to *verify* an assertion. We refer to this as *our* definition. However, we think that in some use cases one might be more interested in finding assumptions required to ensure that the assertion is established in every *concrete* execution. This is an *alternative* definition. These approaches are inherently different. The alternative definition is stronger in the sense that its dependencies are a subset of the dependencies from our definition. Phrased differently, an analysis respecting our definition is also sound with respect to the alternative, but the opposite does not hold.

In this section, we discuss several examples where the expected analysis results differ depending on the definition.

Preconditions of Method and Function Calls

According to our definition, a method call always enforces all preconditions, even though we might rely on a subset of the postcondition which in turn depends on a subset of the precondition. Listing 3.5, page 18, demonstrates this case. Here, the postcondition of `add` does not depend on any precondition. However, calling `add` in `client` requires all preconditions; otherwise, the method call would lead to a verification failure. Thus, the client's assertion depends on the assignments to `a` and `b` in lines 9 and 10. This is precise according to our definition but imprecise according to the alternative definition. Phrased differently, concrete executions that to not establish the precondition of `add` would still satisfy the assertion.

```
1   method add(a: Int, b: Int) returns (res: Int)
2       requires a > 0 && b > 0
3       ensures res == a + b  // (D)
4   {
5       res := a + b          // (D)
6   }
7
8   method client(){
9       var a: Int := 10            // (D)?
10      var b: Int := 10            // (D)?
11      var res: Int := add(a, b)   // (D)
12      assert res == a + b         // (Q)
13  }
```

Listing 3.5: An example illustrating the uncertainties of how to deal with irrelevant preconditions enforced by method calls.

Predicates

Similar to method calls, predicate folds and unfolds can introduce dependencies relevant for verification but not for any concrete execution. To illustrate this, consider Listing 3.6, where a predicate is folded and unfolded later on. The assertion depends only on part of the predicate, namely the permission to `x.f`. However, since the assertion indirectly depends on the `fold` statement, it also depends on the assertion of the entire predicate body. As a consequence, the assignment on line 10 is a dependency. For concrete executions, the assignment does not matter, which is why it is not a dependency with respect to the alternative definition.

```
1  field f: Int
2
3  predicate greater0(x: Ref){
4      acc(x.f) && x.f > 0
5  }
6
7  method predicateClient(x: Ref)
8      requires acc(x.f)      // (D)
9  {
10   x.f := 10           // (D)?
11   fold greater0(x)    // (D)
12   // irrelevant
13   unfold greater0(x)  // (D)
14   assert perm(x.f) > none  // (Q)
15 }
```

Listing 3.6: A Viper program demonstrating how dependencies of predicates are treated differently depending on the definition.

Path-Awareness

Our definition is path-unaware which can lead to unexpected dependencies. Consider Listing 3.7, where assertion (Q) is reachable if and only if $a > 0$. Hence, there cannot be a path where the assumption on line 17, which is reachable if and only if $\neg(a > 0)$, is relevant to prove anything required to reach or verify the assertion. However, our definition considers line 17 to be a dependency. This arises because line 17 is a dependency of line 20, which is a dependency of the assertion. Essentially, the method call on line 20 acts as a synchronization point for dependencies coming from different paths.

```
1  field f: Int
2
3  function retVal(val: Int): Int
4    requires val > 0
5  {
6      val  // (D)
7  }
8
9  method pathUnawarness(a: Int, b: Int)
10   requires b > 0  // (D)?
11 {
12   var res: Int
13
14   if(a > 0){      // (D)
15     res := a      // (D)
16   }else{
17     res := b      // (D)?
18   }
19
20   res := retVal(res + 1) // (D), synchronization point
21
22   if(a > 0){         // (D)
23     assert res > a   // (Q)
24   }
25 }
```

Listing 3.7: A Viper program demonstrating the difference between a path-aware and a path-unaware analysis.

While line 17 is required for *verifying* the the assertion and its dependencies, it is irrelevant for every *concrete* execution reaching the assertion. Thus, for some use cases it would be interesting to take paths into account when computing dependency sets, i.e., to design a path-aware analysis.

Non-Aliasing Proofs

One might encounter dependencies that seem to be unexpected at first but turn out to be needed for non-aliasing proofs. Consider Listing 3.8 where permission to two heap resources is acquired. To verify assertion (Q), it must be checked that the permission amounts of `x.f` and all (potential) aliases add up to exactly 1/2. Non-aliasing of `x` and `y` is guaranteed since the permission amounts to fields `f` would add up to more than 1. To prove this, both preconditions are required which is why they are considered to be dependencies according to our definition.

```
1 field f: Int
2
3 method nonAliasing(x: Ref, y: Ref)
4   requires acc(x.f, 1/2)  // (D)
5   requires acc(y.f)       // (D)
6 {
7   assert perm(x.f) == 1/2 // (Q)
8 }
```

Listing 3.8: A Viper program with a dependency required to prove non-aliasing.

Obviously, removing the second precondition, which inhales permission to `y.f`, is safe which leads to the question whether we should treat this case as imprecision.

Conclusion

These examples, and there might be more, demonstrate that defining dependencies is not trivial and, more importantly, depends on the use case. We argue that our definition is reasonable, among other use cases, to determine why an assertion verifies, to determine which assumptions could be removed, to estimate the impact of removing an assumption on verification, and to compute proof coverage. All of these use cases require the detection of dependencies required for *verifying* assertions. Further, our definition has the convenient property that, given an assertion and its dependencies, we can define a new program that verifies successfully. We exploit this property to implement automated sanity tests in Section 5.3 (Pruning Viper Programs). On the other hand, the alternative definition could be used to estimate what impact removing an assumption has to the correctness of *concrete* executions. This is especially interesting since this information cannot be extracted by simply removing the assumption and re-running verification.

For this thesis, we conservatively chose the weaker definition to guarantee soundness for all discussed use cases. Thus, our analysis is a good foundation but might be too imprecise for some applications.

Chapter 4

Identifying Dependencies in Silicon

Chapter 3 (Definitions) defined what a dependency is in our context and that the goal is to build a dependency graph. From a Viper user's perspective, nodes should correspond to Viper language constructs and edges to direct dependencies between them. This is the Viper dependency graph. Silicon, Viper's symbolic execution backend, does not provide the necessary information to build this graph directly since it operates on lower level assumptions and assertions. For this reason, we introduce an intermediate step which constructs a low-level dependency graph from information collected during verification with Silicon. This low-level graph can be transformed into a Viper dependency graph.

Throughout this chapter, we present an algorithm to compute Viper dependency graphs in the context of Silicon. To this end, we take a top-down approach and start with describing how the Viper dependency graph is assembled when given a low-level dependency graph. Subsequent sections explain how the low-level graph is constructed during symbolic execution. First, we explain how to ensure that all assumptions are tracked. Building on this foundation, we design the algorithm incrementally. We start by extracting direct dependencies, which are further categorized into dependencies extracted from the UNSAT core and permission flow. Then, we clarify how these steps already deal with branches and loops. Special treatment is required, however, for correctly detecting dependencies to control-flow conditions. Finally, we discuss how to find dependencies across program components.

Note that in this chapter we assume that the Viper program at hand verifies successfully. Further, examples given in this chapter are partly simplified to hide details specific to Silicon when they are irrelevant for the discussion. Importantly, while a Viper variable x is mapped to an internal variable such as x@01@08, we omit this detail by using x for both levels. When necessary, we introduce different versions of a variable denoted by, e.g., $x1$ and $x2$.

4.1 Assembling the Viper Dependency Graph

Although assembling the Viper dependency graph is the last step of the analysis, we explain it first because it is the foundation for understanding low-level dependencies. In this section, we assume that we are already given a low-level dependency graph consisting of one node per assumption and assertion encountered during symbolic execution and edges that indicate direct dependencies between them, as illustrated by Figure 4.1a, page 25.

Computing the corresponding Viper dependency graph, consisting of Viper nodes (Definition 3.2, page 11) and direct dependencies between them (Definition 3.5, page 13), is done by using the following mappings from Viper nodes to low-level nodes.

Definition 4.1 (Viper Assumption)
assumptions(S) *denotes the set of assumptions introduced during symbolic execution of Viper node* S *on any path. If* *assumptions*$(S) \neq \emptyset$, S *can act as a Viper assumption.*

Definition 4.2 (Viper Assertion)
assertions(S) *denotes the set of assertions required by Viper node* S *on any path. If* *assertions*$(S) \neq \emptyset$, S *can act as a Viper assertion.*

For detailed definitions of assumptions and assertions introduced by a Viper node, we refer to Schwerhoff [31], in particular to Figure 3.6 on pages 60 and 61, where assumptions are introduced by calling the *produce* function and assertions by calling the *assert* and *consume* functions. Examples were already discussed in Section 3.2 (Assumptions and Assertions), particularly in Listing 3.1, page 13.

We can also define the inverse mapping, namely from low-level nodes to Viper nodes.

Definition 4.3 (Source)
source(a) *denotes the Viper node that introduced assumption or enforced assertion* a*.*

$$\mathit{source}(a) = V \iff a \in \mathit{assumptions}(V) \cup \mathit{assertions}(V)$$

We use the term "source" to emphasize that the low-level node was created while symbolically executing its source, i.e., the corresponding Viper node. Notice that there is a one-to-many mapping from one Viper node to multiple low-level nodes. Therefore, source(a) is uniquely defined for each a and is stored directly in the low-level node representing a.

The Viper dependency graph can now be computed by merging low-level nodes with identical source and preserving all dependencies while doing so. The resulting graph consists of Viper nodes and direct dependencies between them. The merge operation is defined as follows.

Definition 4.4 (Graph Conversion)
Given a low-level dependency graph with nodes V_L and edges E_L, the corresponding Viper dependency graph $G_V = (V_V, E_V)$ is defined by

- $V_V = \bigcup_{v \in V_L} source(v)$ *and*
- E_V *such that*

$$\forall s_V, t_V \in V_V.\ (s_V, t_V) \in E_V$$

$$\Longleftrightarrow$$

$$(\exists s_L, t_L \in V_L.\ (s_L, t_L) \in E_L\ \wedge\ source(s_L) = s_V\ \wedge\ source(t_L) = t_V)$$

Let us illustrate this on an example. Figure 4.1a visualizes a low-level dependency graph. The underlying program is described by the headings when read from left to right. These headings also correspond to the source associated with each node inside the box. Colors indicate different node types, namely assumptions shown in blue, chunk creations in green, and assertions in red. The graph resulting from merging nodes with identical source is presented in Figure 4.1b. As can be seen, we are left with one node per box, i.e., one node per source. This is the Viper dependency graph as defined in Section 3.1 (Dependency Graph).

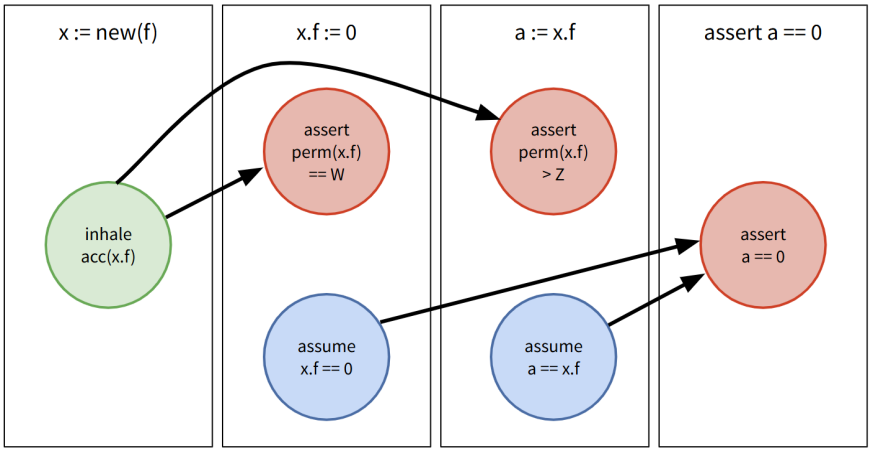

Figure 4.1a: A low-level dependency graph as constructed in later sections. Details about chunks and the field assignment encoding are not relevant and thus simplified.

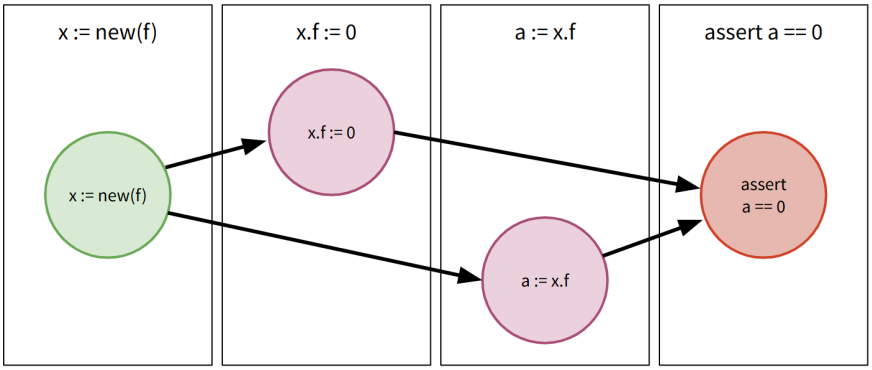

Figure 4.1b: The Viper dependency graph corresponding to Figure 4.1a.

Merging nodes has an important side effect. A Viper node v_L can be both an assumption and an assertion at the same time, namely when being a result of merging low-level assumption and assertion nodes. We emphasize this by using purple nodes, e.g., for `x.f := 0` and `a := x.f` in Figure 4.1b. As a consequence, indirect dependencies can be detected by computing transitive closures; for example, the transitive closure of `assert a == 0` contains the initialization `x := new(f)`. This indirect dependency did not exist in the low-level graph.

In conclusion, merging nodes with identical source serves two purposes. First, it hides unnecessary low-level details from users making the dependency graph more compact and user-readable. Second, it enables the computation of transitive dependencies. In the following sections, we explore the construction of the low-level dependency graph by detecting direct dependencies between low-level assumptions and assertions.

4.2 Collecting Assumptions

We have seen how to translate a low-level graph into a Viper graph. Next, we discuss how to the low-level dependency graph itself is constructed. As a foundation, we need to collect all assumptions added to the proof context throughout symbolic execution. These nodes are referred back to when determining the dependencies of an assertion. Remember that Silicon's symbolic state consists of the path condition, the store, and the symbolic heap. We discuss each of these data structures separately. Then, we present the information stored in each node.

Path Conditions

The path condition keeps track of constraints learned on the current path. All of these are assumptions. We can intercept them before they are added to the path condition and create a node accordingly. Note that our graph contains nodes created on various paths encountered during symbolic execution. This is a major difference to the path condition maintained as part of the symbolic state which is subject to one specific path.

Simplified Store Model

The store maintains a map from program variables to internal terms, e.g., `x -> 1`. The fact that a program variable has a certain value is an assumption that needs to be tracked in our graph. It turns out that dealing with such assumptions encoded in the store is hard. We exemplify this briefly and then propose a simplified store model which externalizes such assumptions to the path condition, thus, simplifying dependency analysis.

Symbolically executing assignments results in updating the map for the assigned program variable, e.g., `x := 0` updates the store such that `x -> 0`. Subsequently, an assertion like `x == 0` evaluates to `0 == 0` as a direct consequence of looking up `x`'s value in the store. Silicon simplifies this term to `true` and concludes that the assertion is established. Tracking the dependencies of this expression, e.g., to assignment `x := 0`, is difficult because it might involve tracking which variables were looked up in the store and information about the latest statement that updated this program variable. Another challenge is the need to determine whether the looked-up term even impacts the assertion.

To overcome these difficulties, we implement all updates to the store by creating fresh, internal variables, updating the store, and adding an equality to the path condition[1]. For instance, the assignment `x := 0` updates the store

[1]Silicon already implements this process for most but not all updates. In particular, if the right-hand side of the assertion is a concrete value or a single variable, Silicon usually

such that `x -> x1` and the assumption `x1 == 0` is added to the path condition. An assertion like `x == 0` now evaluates to `x1 == 0` and cannot be simplified further. Instead, the path condition is consulted to retrieve the previously added assumption `x1 == 0` that proves the assertion. In essence, all constraints on values of program variables are stored in the path condition. Therefore, whenever the current value of a program variable is relevant, e.g., when asserting `x == 0`, the path condition has to be taken into account. As discussed earlier, tracking assumptions in the path condition is straightforward and already covered.

By enforcing the creation of fresh, internal variables on all updates to the store, dependency analysis can ignore all mappings in the store. Similarly, updates to values and permissions stored in heap chunks are forced to create fresh, internal variables such that constraints on their values are stored in the path condition. Figure 4.2 visualizes some assumption nodes as created when using the simplified store model. Notably, permissions are encoded as fractionals and are hence added to the path condition as indicated by the blue assumption nodes.

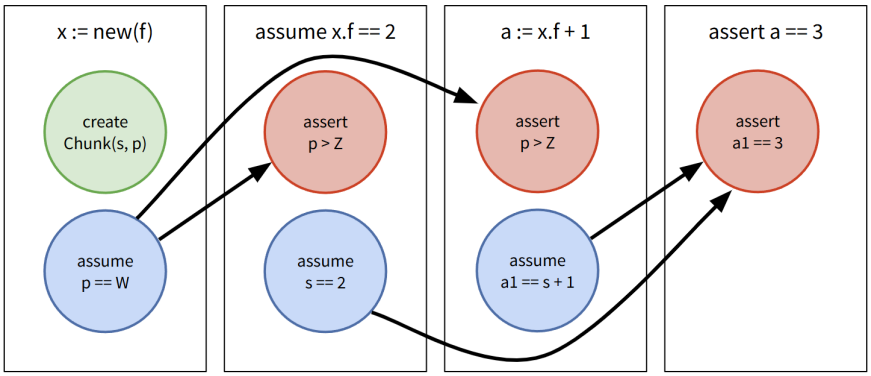

Figure 4.2: The low-level assumption nodes as created when using the simplified store model. Creation of the chunk for x.f implicitly creates fresh, internal variables s and p representing the value and permission amount associated with `x.f`, respectively. Similarly, the assignment `a := x.f + 1` creates the internal variable a1. Assertions and dependencies are grayed out and some implementation details related to field assignments are omitted.

To conclude, with our proposed changes, all assumptions introduced during verification are encoded in the path condition.

Symbolic Heap

The symbolic heap keeps track of all available heap locations at any point during symbolic execution by maintaining a map from heap references to

updates the store directly without creating a fresh, internal variable.

immutable chunks. The fact that a given chunk exists is to be understood as an assumption. Due to the simplified store model, it is already tracked by the constraints on the value and permission amount associated with the chunk, both of which are added to the path condition. Additionally, we introduce so-called "create" nodes representing the inhaling of a chunk, as indicated by the green node in Figure 4.2, page 27. The motivation behind this is to explicitly track permission flow to enable corresponding visualizations.

Information Stored in Assumption Nodes

Previously, we addressed how assumptions are tracked. For each of them an assumption node is created with the following information attached to it. First, each node is associated with a *node type* such as assumption or assertion and stores the corresponding assumption or assertion. Moreover, it is associated with a unique identifier. As explained in Section 4.1 (Assembling the Viper Dependency Graph), the source of the node has to be determined. To this end, the symbolic state keeps track of the currently executed Viper node and attaches it as a source to all low-level nodes. This Viper node not only stores the corresponding Viper expression or statement but also its exact position in the source code. This position is crucial for uniquely identifying the statement or expression, e.g., when the source code contains several, identical statements.

To summarize, collecting assumptions as described above results in a graph with one node per assumption made throughout symbolic execution of any path. Subsequently, we add assertion nodes including edges to assumption nodes representing the direct dependencies.

4.3 Extracting Dependencies

Assertions requiring assumptions stored in the path condition are verified by the SMT solver. These assertions are intercepted to create corresponding assert nodes. To extract their dependencies and add edges accordingly, we rely on the SMT solver and UNSAT cores, which were briefly addressed in Section 2.4 (SMT Solver and UNSAT Cores).

More precisely, an assertion verifies successfully if and only if the SMT solver can prove unsatisfiability of its negation. In that case, the UNSAT core contains all assumptions used for the proof. We exploit this information to add edges to our graph accordingly.

Note that we assume the SMT solver is sound and the UNSAT core is complete, i.e., the reported dependencies are sufficient to prove the assertion. On the other hand, minimality and uniqueness of UNSAT cores are not required for a sound analysis, as briefly addressed in Section 3.3 (Dependency).

4.4 Permission Flow

In this section, we describe how assertions enforcing properties on the symbolic heap are taken into account. The symbolic heap stores information about the available heap resources. They are created, modified, and removed throughout symbolic execution via operations triggering the inhale or exhale of permissions to resources.

Let us start by reasoning about all kinds of assertions concerning the symbolic heap. First, assertions might enforce conditions on the value or permission amount associated with heap locations. The relevant assumptions required to prove such assertions are stored in the path condition due to the simplified store model introduced in Section 4.2 (Collecting Assumptions). Hence, such assertions are verified by the SMT solver and already covered as described in Section 4.3 (Extracting Dependencies). Second, the values and permission amounts of heap locations can be modified and heap locations can be exhaled fully. These operations are implemented by adding and removing immutable chunks. Section 4.2 (Collecting Assumptions) already addresses the creation of chunks, which is considered to be an assumption. Removal of a chunk is always preceded by asserting that this chunk satisfies the permission requirements which is already covered by Section 4.3 (Extracting Dependencies). Therefore, these dependencies are already detected.

However, to distinguish between merely asserting permission requirements and removing chunks, so-called “remove” nodes are created to represent the removal of chunks. Similarly, “create” nodes are added to indicate the creation of a new chunk. These nodes are introduced by statements inhaling or exhaling permission but also the ones modifying heap locations, for example, field assignments. The main motivation behind creating such nodes is to allow visualization of permission flow, i.e. the graph can be filtered for create and remove nodes and dependencies between them.

To keep visualizations compact and readable, we usually show chunk creations as one node representing the chunk itself *and* the assumption about its permission amount. Similarly, we present the removal of a chunk and the preceding assertion of permission requirements as one remove node, as depicted in Figure 4.3.

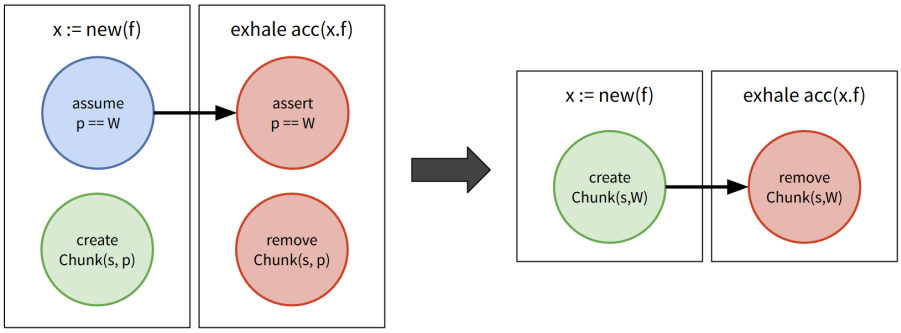

Figure 4.3: The low-level nodes created when inhaling and exhaling a chunk, respectively, on the left-hand side. The variables s and p denote the value and permission amount of the chunk, respectively. In the remainder of this thesis, this details are simplified as depicted on the right-hand side.

As a side note, our implementation explicitly adds edges between create and remove nodes as well as edges connecting them to assume and assert nodes. This has historic reasons since we implemented permission flow before introducing the simplified store model. Without this model, permissions are not store in the path condition, therefore requiring a different technique to find dependencies for assertions on the symbolic heap. However, we think that these edges have become obsolete when introducing the simplified store model but we did not yet explore this and leave simplification for future work. For this reason, explanations of which edges are added by our implementation have been externalized to the appendix, Section 10 (Explicitly Tracking Permission Flow).

Impure method calls, predicates, and magic wands are internally implemented as exhaling and inhaling permission and are thus already taken into account. Section 10 (Explicitly Tracking Permission Flow) in the appendix briefly addresses these cases to make this more clear.

4.5 Branches and Loops

In previous sections, we covered various kinds of dependencies but we ignored an important detail concerning symbolic execution: it explores each (feasible) program path separately. The Viper dependency graph should consider all paths. Thanks to how we merge nodes, see Section 4.1 (Assembling the Viper Dependency Graph), this is already correctly handled by the dependency analysis. For the sake of completeness, this section demonstrates how dependencies encountered on different paths are aggregated. Further, we clarify how loops and invariants are correctly dealt with.

Aggregating Dependencies Detected on Different Paths

Sections 4.1 through 4.4 provide all techniques required to find and aggregate dependencies from different program paths. We demonstrate this based on the example in Figure 4.4a, page 32, which contains the low-level dependency graph of a program containing a branch. Symbolic execution explores both branches separately by forking execution as indicated by the orange line. On the first path, depicted above the orange line, branch condition `c >= 0` is assumed, and the if-branch as well as the program after the branch are executed. In particular, the assertion is verified, which creates an assertion node and dependencies as discussed in previous sections. Similarly, the second path, below the orange line, is executed, also creating a node and dependencies for the assertion.

This example shows that a single Viper assertion can have several assertion nodes, possibly encountered on different paths. The same holds for assumptions, as shown for the assignment to `d` in Figure 4.4a. Since these nodes have the same source, they are merged when assembling the Viper dependency graph provided in Figure 4.4b. Here, we only have one node for the assertion and the assignment, respectively. As a consequence, the Viper dependency graph does not encode path information, making it path-unaware. This is not an imprecision according to our definition as discussed in Section 3.5 (Discussion and Refined Definitions).

```
1    var a: Int, b: Int, c: Int, d: Int
2    assume a > 10
3    if(c >= 0){
4        b := c
5    }else{
6        b := a
7    }
8    assert b > 0
9    d := b + c
```

Listing 4.1: A Viper program with a branch.

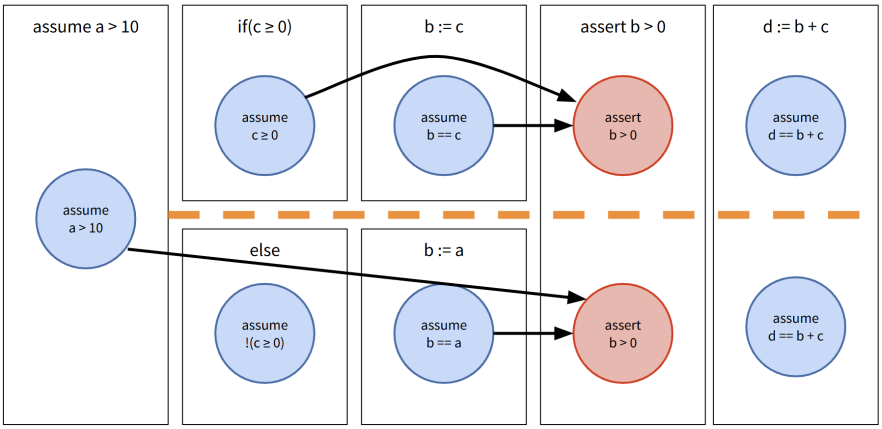

Figure 4.4a: The low-level dependency graph of the Viper program given in Listing 4.1.

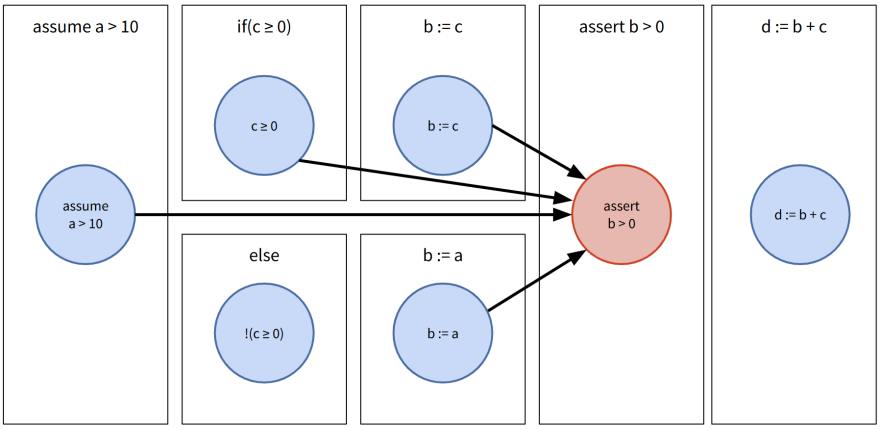

Figure 4.4b: The Viper dependency graph of the Viper program given in Listing 4.1.

Loops and Invariants

Loops are internally encoded using `inhale` and `exhale` statements as well as branches, see Schwerhoff [31] for details. These operations are already taken into account by the dependency analysis, i.e., it already detects dependencies

in the presence of loops. Nevertheless, we present an example here for the purpose of clarification and demonstration.

Figure 4.6a, page 34, illustrates the low-level dependency graph of the program in Listing 4.2 containing a loop. Symbolic execution verifies loops in three steps as if there were three branches, separated by the orange lines. The branch on the left-hand side verifies that the invariants are established before the loop. In the middle branch, it is ensured that the loop body preserves all invariants. On the right-hand side, the invariants and the negation of the loop condition are assumed and verification continues after the loop. The dependencies in each of the three parts are discovered as discussed in the previous sections. Notice how each invariant introduces several assumptions and assertions. Moreover, the graph does not contain any loops, which is due to the fact that symbolic execution is purely sequential.

```
1  var a: Int := 0
2  var i: Int := 10
3  while(i > 0)
4    invariant i >= 0
5    invariant a == 5*(10-i)
6  {
7    i := i - 1
8    a := a + 5
9  }
10 assert a == 50
```

Listing 4.2: A Viper program with a loop whose depenency graphs are visualized in Figure 4.6a and Figure 4.6b.

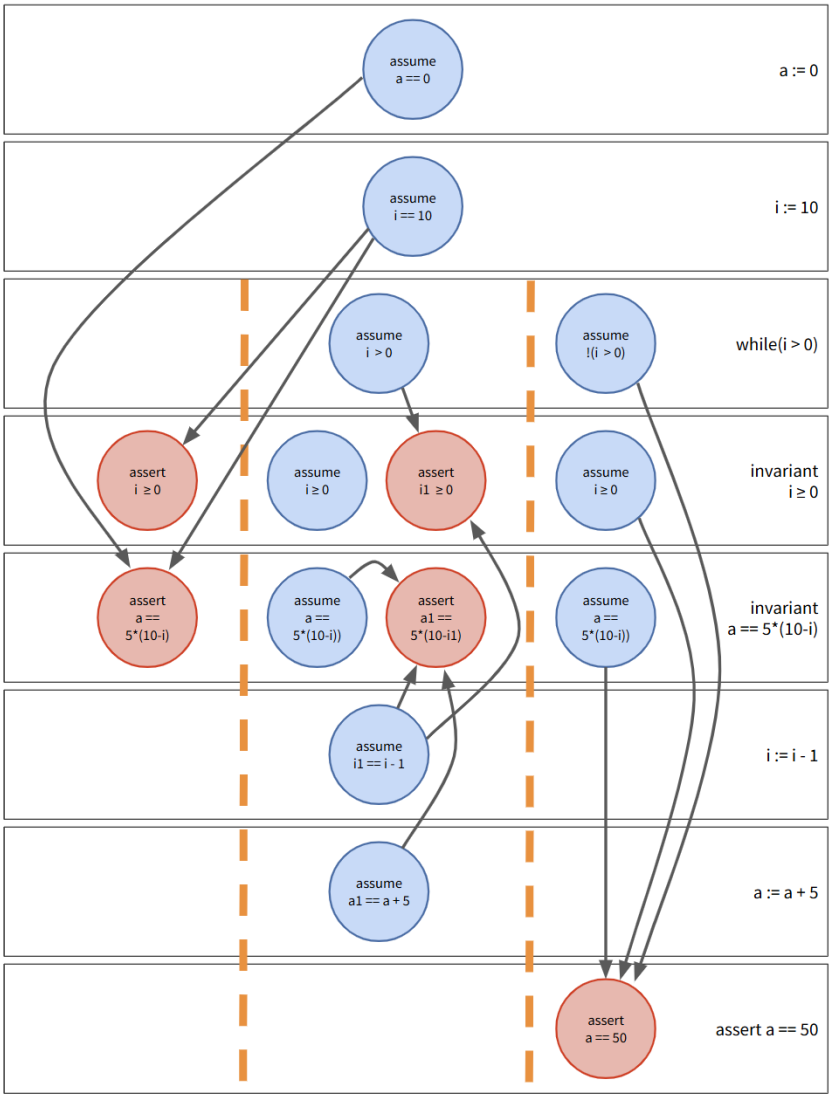

Figure 4.6a: The low-level dependency graph associated with Listing 4.2.

The corresponding Viper dependency graph is provided in Figure 4.6b. Importantly, the low-level nodes of each invariant are merged into a single Viper node. This is key to compute transitive dependencies, e.g., to reveal the indirect dependency from the first assignment, `a := 0`, to the assertion. Dependencies from statements in the loop body to the assertion after the loop can be detected as well. Note that, in contrast to the low-level graph, the Viper dependency graph contains loops.

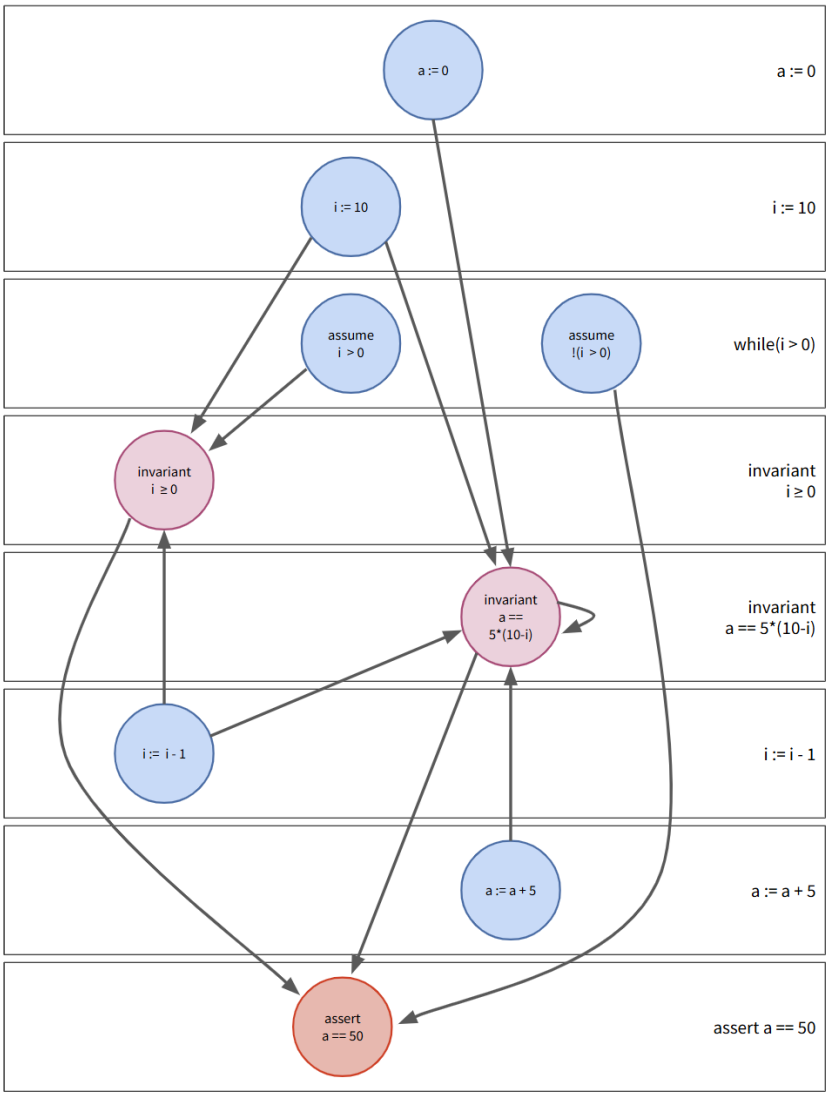

Figure 4.6b: The Viper dependency graph associated with Listing 4.2.

4.6 Dependencies to Control Flow Conditions

The previous section explored how dependencies encountered on different paths are aggregated. What we did not cover is how to detect dependencies to branch and loop conditions. One might think that those are already covered by the dependency analysis since these assumptions are stored in the path condition. However, it turns out that some dependencies are missed in the presence of infeasible paths and redundant branch conditions. More concretely, a branch condition can be relevant for verifying an assertion when (1) it introduces relevant, previously unknown constraints, or (2) it introduces

relevant but redundant constraints, or (3) it makes the branch infeasible trivially proving the assertion. For case (1), the dependency is detected using the techniques discussed in Section 4.3 (Extracting Dependencies). However, cases (2) and (3) are not yet covered. Subsequently, we provide two different examples illustrating them both. Then, we present a solution that resolves both cases at once. The key observation is that some dependencies are missed due to infeasible paths not being executed. Consequently, the solution is based on exploring infeasible paths and adding dependencies to the proof of infeasibility.

Unreachable Assertions

Consider Listing 4.3 which contains an unreachable `assert false` statement. So far, symbolic execution would detect infeasibility of the branch, since the branch condition contradicts the facts learned from the assignment, and hence never executes it. As a consequence, the assertion is never reached and analysis does not even create an assertion node, let alone detect its dependencies. From a user's perspective it is important to understand why a statement is unreachable. In other words, the proof of infeasibility should be reported as a dependency of the assertion, as suggested in Listing 4.3.

```
1 var a: Int := 0 // (D)
2 if(a > 0){      // (D)
3   assert false  // (Q)
4 }
```

Listing 4.3: All expected dependencies of an unreachable assertion.

Redundant Branch Conditions

The previously made observation can be generalized: The fact that an assertion is unreachable on *some* paths might be important information. We demonstrate an example where this matters in Listing 4.4 which contains two sequential branches with identical branch condition. We assume that the omitted, irrelevant code fragments do not modify `a` and always terminate.

The expected result is that assignment `c := b/a`, which requires `a != 0`, depends on branch condition `a > 0` on line 6. However, the dependency analysis as discussed so far would report a dependency to branch condition `a > 0` on *line 2* instead! This is unexpected and wrong. To understand why this happens, we need to look at how branching is handled in Silicon.

```
1 var a: Int, b: Int, c: Int
2 if(a > 0){
3   // irrelevant
4 }
5 // irrelevant
6 if(a > 0){    // (D)
7   c := b / a  // (Q)
8 }
```

Listing 4.4: All expected dependencies in a Viper program with redundant branch conditions.

The subsequent explanations are supported by Figure 4.7. When symbolic execution arrives at the first branch, it forks execution resulting in two paths with conditions `a > 0` and `!(a > 0)`, respectively. The paths are executed separately as indicated by the orange line. Eventually, both are forked again when arriving at the second branch, resulting in a total of four paths. Two of these paths are skipped, indicated by the red line, because they are infeasible. The other paths encounter redundant assumptions as indicated by the gray nodes. Overall, only one of four paths executes the assignment `c := b/a`. To assert `a != 0` on this path, either of the branch conditions (line 2 or line 6) can be used as a justification. In practice, we observed that the dependency analysis reports *line 2* as a dependency which explains the unsound result for this example.

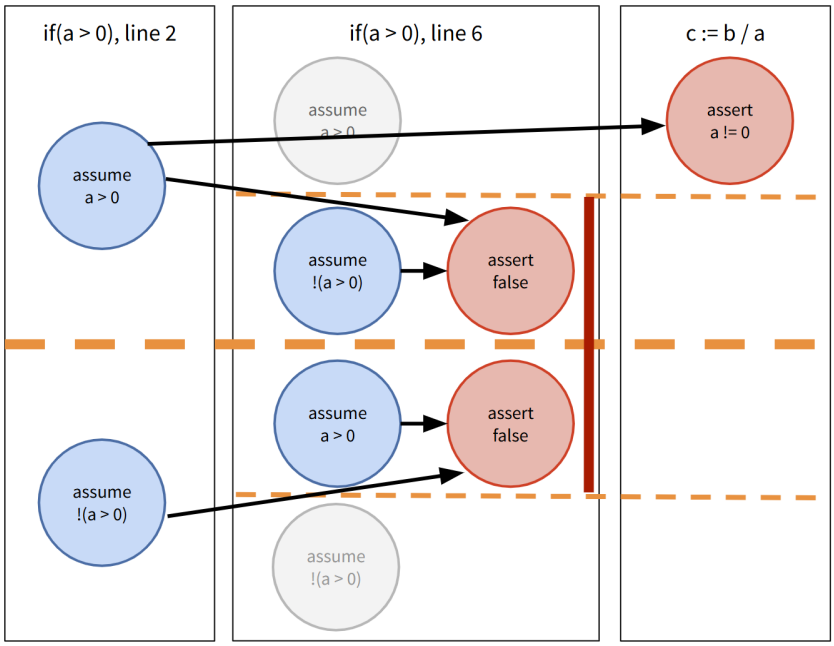

Figure 4.7: The low-level dependency graph associated with the program in Listing 4.4 illustrating how the dependency to a redundant branch condition is missed if not handled carefully.

The underlying issue is that the redundant branch condition is ignored. Solving it poses two challenges: First, since branches can be interleaved arbitrarily, it is not clear how to find the relevant branch condition. Second, even if we could point out the relevant branch condition, the SMT solver might still report another assumption as a dependency instead. Our solution avoids these two challenges by exploiting infeasible branches to detect the missing dependencies as discussed next.

Infeasibility Proofs and Nodes

So far, we looked at two concrete examples complicating the detection of dependencies to control flow conditions. Both cases can be solved by adding dependencies to infeasibility proofs. We already argued that the infeasibility proof explains why an unreachable assertion verifies.

Why this also works for redundant branch conditions is more subtle. It follows from the fact that a redundant branch condition implies the existence of an infeasible path. Formally speaking, given redundant branch conditions a, b such that $b \Rightarrow a$, $\neg a \wedge b$ is a contradiction, i.e., results in an infeasible path. In Listing 4.4, page 37, a corresponds to the first and b to the second branch condition. The proof of infeasibility depends on both (redundant) branch

conditions $\neg a$ and b[2], i.e., infeasibility was introduced by this branch. Hence, reporting the proof of infeasibility as a dependency solves the problem. We illustrate this in Figure 4.8 where assertion nodes and edges to the proof of infeasibility were added as a result of executing infeasible paths.

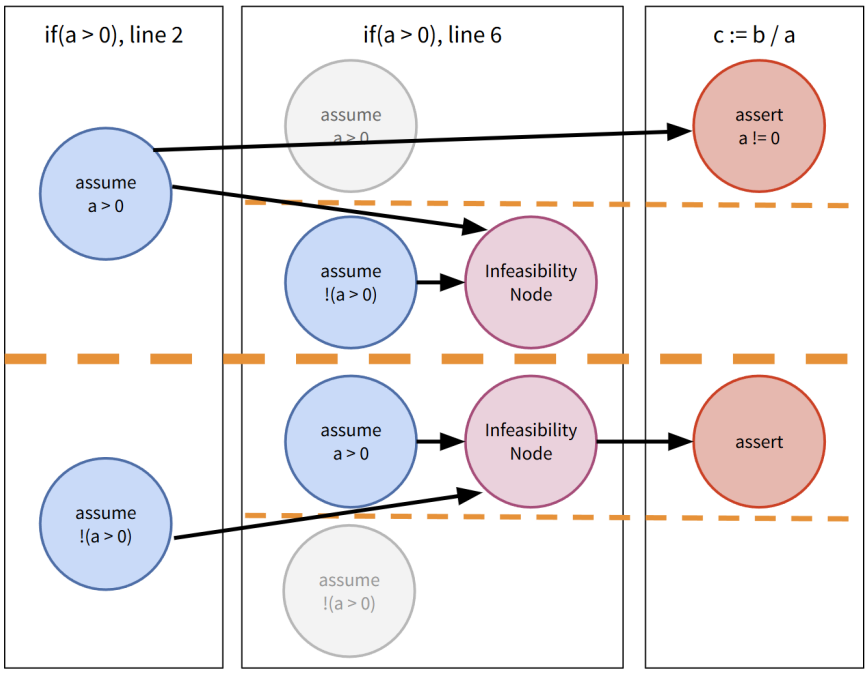

Figure 4.8: The low-level dependency graph associated with Listing 4.4 demonstrating how infeasibility proofs help detecting dependencies to branch conditions.

Hence, to solve these two cases, we need to (1) extract the dependencies required for the infeasibility proof, i.e., asserting `false`, and (2) add them to all relevant assertions. Step (1) was already covered in previous sections. For step (2), we need to clarify what the relevant assertions are. In theory, those are the assertions that depend on the branch condition in the sense that removing this condition would lead to a verification failure. It is not clear how these can be identified; for example, not all unreachable statements depend on the infeasibility proof as demonstrated in Listing 4.5.

[2]That is under the assumption that the path was previously feasible. If the path was already infeasible, the branch condition is irrelevant, which is why we can ignore this case.

```
1 var a: Int := 0
2 if(a > 0){
3   var b: Int := 0 // (D)
4   assert b == 0   // (Q)
5 }
```

Listing 4.5: All expected dependencies of an unreachable assertion that does not depend on the infeasibility proof consisting of lines 1 and 2.

We bypass these difficulties by implementing a sound yet imprecise solution. While determining the relevant assertions is challenging, finding all candidates is simple since it amounts to *all* assertions encountered on the infeasible path. To add the dependencies, we create a so-called infeasibility node with dependencies to the assumptions used for the infeasibility proof whenever detecting the infeasibility of a path. Symbolic execution is then forced to continue and explore the path to the end. Note that we can step through infeasible path without (symbolically) executing anything since all assertions are guaranteed to verify trivially. Accessing or updating the proof context as well as invoking the SMT solver is obsolete. Instead, it is sufficient to create assertion nodes with a dependency to the infeasibility node. This might add irrelevant dependencies to the graph which is sound, albeit not precise. As a result, we get the dependency graphs illustrated in Figure 4.8, page 39, for the redundant branch conditions and Figure 4.9 for an unreachable assertion.

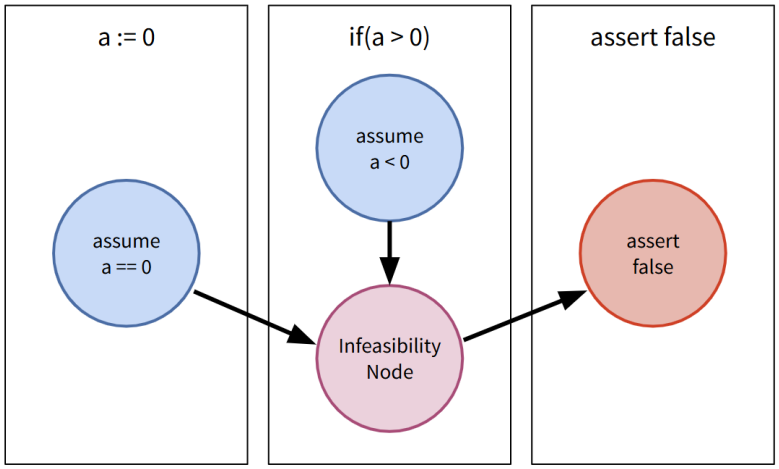

Figure 4.9: The low-level dependency graph associated with Listing 4.3, page 36, demonstrating how infeasibility nodes are used as dependencies for otherwise unreachable assertions.

Conclusion

In short, we reliably detect dependencies to control flow conditions by executing infeasible paths and reporting the proof of infeasibility to all assertions encountered while doing so. This was the last step necessary to ensure sound dependency analysis in the modular case. That is, we now detect all dependency *within* program components. The next section goes beyond this by detecting dependencies *across* program components.

4.7 Dependencies across Program Components

In this section, we add dependencies across components. To this end, we first describe how to detect dependencies to domains and functions. Then, dependencies across methods, which are introduced by method calls, are discussed.

Domain and Function Axioms

We should detect and report when a domain axiom[3], a function postcondition, or the body of a function is used to verify an assertion. In Silicon, all of the above are encoded as axioms which are passed to the SMT solver before verification of any component starts. As a result, the SMT solver knows about these assumptions and can already detect dependencies to them. All we need to do is create a node for each axiom such that dependencies can be detected analogously to Section 4.3 (Extracting Dependencies).

Method Postconditions

Dependencies between methods are introduced by method calls and, more concretely, by the fact that the caller assumes the callee's postconditions. In Silicon, method postconditions are not encoded as axioms and thus require a different technique. Instead, the postconditions are retrieved from the callee's definition and inhaled when symbolically executing a method call. Moreover, it is not guaranteed, for example, that a method foo has already been verified when verifying a client method which calls foo. As a result, the graph might not yet contain any node representing foo's postconditions. In fact, each method creates its own low-level dependency graph which is disconnected from other methods' graph. We need to ensure that these graphs can be joined in order to detect dependencies between them.

To achieve this, nodes created by the caller which represent the inhale of a postcondition store information on how to perform the join. For this purpose,

[3]Note that domain functions do not need to be considered since their functionality is defined through the domain axioms.

we use the reference to the postcondition as retrieved from the callee's definition during verification. Note that such nodes store the method call as their source and now additionally store a reference to the postcondition. On the other end, the source stored in nodes created by the callee while verifying the postcondition is equal to this postcondition. Together, the information stored in the caller's and callee's nodes is sufficient to join the graphs, as illustrated in Figure 4.10 where red arrows represent the inter-method dependencies added by the join operation. As can be seen, `client`'s assertion now indirectly depends on parts of `foo`'s body and potentially the precondition, as suggested by loose arrows.

```
1  method foo(a: Int) returns (res: Int)
2      requires a > 0
3      ensures res > 0
4      ensures res < 50
5  {
6      res := 1
7      if(a < 50){
8          res := a
9      }
10 }
11
12 method client()
13 {
14     var a: Int := 10
15     var res: Int := 0
16     res := foo(a)
17     assert res >= 0
18 }
19 }
```

Listing 4.6: A Viper program with a client calling method `foo`.

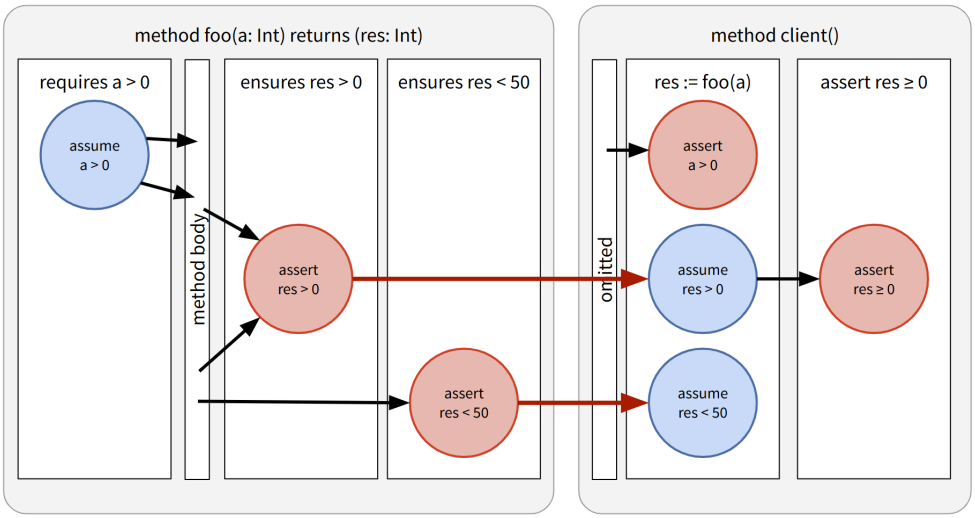

Figure 4.10: The low-level dependency graph of the program in Listing 4.6 illustrating how graphs created by different methods are joined via method calls and postconditions. Details of the method bodies are omitted.

Joining graphs by preconditions is not necessary because all nodes created by a method call are merged anyway. Thus, the assumption of postconditions already depends on everything required to prove the preconditions of a method call. In essence, a method call assumes that each postcondition depends on each precondition, as briefly discussed in Section 3.5 (Discussion and Refined Definitions).

Abstract Functions and Methods

Abstract functions and methods constitute a special case since they do not have bodies. With the previous techniques, dependencies to the postconditions of such functions and methods can already be detected. However, since they do not have a body, these postconditions do not depend on anything themselves. We assume that each postcondition of an abstract function or method depends on each precondition. The corresponding edges are explicitly added when verifying the well-formedness of such functions and methods.

4.8 Summary

To summarize, we describe the dependency analysis once again, this time using a bottom-up approach. During verification, the low-level dependency graph is built. Direct dependencies are extracted from UNSAT cores provided by the SMT solver. Dependencies to control-flow conditions are detected by enforcing the execution of *all* paths, whether feasible or not, and by

creating dedicated infeasibility nodes. To support reasoning across program components, we need to consider domain and function axioms and join graphs created by different methods via method calls and postconditions. Branches and loops are already trivially supported.

The result of the described algorithm is a low-level dependency graph with (potentially) many nodes per source, e.g., per Viper statement. As a final step, this graph is lifted to a Viper dependency graph by merging nodes belonging to the same source, i.e., introduced by the same Viper node. The final Viper dependency graph is a compact and user-friendly representation of the dependencies present in Viper programs.

Chapter 5

Applications of Viper Dependency Graphs

Chapter 4 (Identifying Dependencies in Silicon) covered the construction of the Viper dependency graph of a given program. In this section, we describe information encoded in the dependency graph and how to extract them to gain insights about the underlying program. First, we introduce assumption types which can be used by graph tools to filter for certain assumptions or assertions. Further, we explore two direct applications of dependency graphs, namely proof coverage and pruning of Viper programs.

5.1 Assumption and Assertion Types

So far, we have treated each assumption as equally important. However, in some uses case one might be interested in a subset of assumptions; for example, in unverified assumptions which were explicitly added to the program. To account for various use cases, we associate each dependency node with an assumption or assertion type. This enables filtering analysis results for specific types giving users the option to remove irrelevant information and, thus, making analysis more user-friendly.

Most importantly, we distinguish between explicit, implicit, and internal assumptions. Broadly speaking, during verification, explicit assumptions are assumed to hold without further justification.

Definition 5.1 (Explicit Assumption)
An assumption is explicit iff it is introduced due to a program component's precondition, an `inhale` *or* `assume` *statement, or a postcondition of an unverified method or function.*

In contrast to that, implicit assumptions are justified by either being part of the runtime code or a preceding assertion; for example, the assumption

introduced by a `fold` statement is justified by asserting the predicate's body.

Definition 5.2 (Implicit Assumption)
An assumption is implicit iff it is introduced by an invariant, verified postcondition, domain axiom, `fold`, `unfold`, `apply`, *or* `package` *statement, or it reflects the semantics of a runtime statement, namely an assignment (including* `new` *expressions), branch or loop condition as well as method or function call.*

Lastly, we refer to all other assumptions as internal assumptions. They are introduced due to implementation details such as state consolidation or heap summary.

Definition 5.3 (Internal Assumption)
An assumption is internal iff it is neither an explicit nor an implicit assumption.

As a side note, our implementation classifies these assumption types in a more fine-grained manner. In particular, implicit assumptions are split into path conditions, loop invariants, rewrites, axioms, and unclassified assumptions. This allows for more fine-grained filtering in graph queries.

Analogously, we distinguish between explicit and implicit assertions.

Definition 5.4 (Explicit Assertion)
An assertion is explicit iff it is introduced by a postcondition, an `assert` *statement, or an* `exhale` *statement.*

Definition 5.5 (Implicit Assertion)
An assertion is implicit iff it is not an explicit assertion.

For example, ensuring that permission requirements are satisfied or a divisor is not 0 as part of an assignment are both implicit assertions.

As mentioned briefly, we associate each dependency node with an assumption or assertion type when creating it. The type mostly depends on the Viper statement that introduces it, i.e., the node's source. However, there are exceptions, for example, when deliberately marking some nodes as internal in order to hide implementation details. Assumption and assertion types are relevant for graph queries and the computation of proof coverage, which is addressed in the next section.

5.2 Proof Coverage

Apart from revealing dependencies, dependency graphs can be used to extract additional metrics, one of which is proof coverage. Based on the dependency graph, we can compute the fraction of nodes that are dependencies of a given assertion. We say that these nodes are *covered* by the assertion. As already motivated in Section 1 (Introduction), proof coverage might be used as an

indication of verification progress, similar to test coverage which, estimates the thoroughness of a test suite.

We propose to restrict proof coverage to a single program component. In many cases, this restriction is reasonable since the proof coverage of an assertion in one method should not be affected by assumptions made in another, especially if that method does not even call the other one. While defining and computing proof coverage over entire programs would be possible, we did not explore this further. For this thesis, we use the following definition of proof coverage. Some examples are provided in Listing 5.1.

Definition 5.6 (Proof Coverage of Assertions)
Proof coverage of a set of Viper assertions Q with respect to program component P is defined by

$$\rho(Q, P) := \frac{|D \cap A|}{|A|}$$

where D is the minimal Viper dependency set of Q, i.e., $D \models_{min} Q$, and A is the set of all Viper assumptions in P.

Proof coverage is defined on the *minimal* Viper dependency set and thus requires a *precise* dependency analysis. As a consequence, computing this metric using the dependency graph resulting from our imprecise analysis algorithm can only give an approximation of proof coverage. Further, it is important to emphasize that proof coverage is computed based on the Viper dependency graph, as opposed to the low-level graph.

```
1  // method proof coverage=2/3
2  method incr(a: Int) returns (res: Int)
3      requires a >= 0   // (D)
4      requires a < 10
5      ensures res > 0   // (Q), proof coverage=2/3
6  {
7      res := a + 1      // (D)
8  }
9
10 // method proof coverage=1
11 method foo(n: Int, res: Ref)
12 {
13     var a: Int := 1
14     var b: Bool
15     if(b){          // (D)
16         a := incr(a)
17     }else{
18         b := true  // (D)
19     }
20
21     assert a > 0       // proof coverage = 2/4
22     assert b == true   // (Q), proof coverage = 2/4
23 }
```

Listing 5.1: A Viper program exemplifying proof coverage.

Next, we define proof coverage with respect to a method. To this end, we propose considering explicit assertions and their dependencies. Some examples are provided in Listing 5.1.

Definition 5.7 (Proof Coverage of Methods)
Proof coverage of method P is defined by

$$\rho(P) := \rho(Q_P, P)$$

where Q_P is the set of explicit assertions in P.

An alternative would be to define proof coverage as the fraction of assumptions covered by the *postconditions*. While we did not explore this metric, the dependency graph already contains all information to do so.

In brief, we leverage the information encoded in the dependency graph to compute proof coverage. This metric provides insightful information about verification progress, partly also by revealing uncovered nodes.

5.3 Pruning Viper Programs

By definition, the dependency graph reveals all (direct and indirect) dependencies of all assertions present in the underlying program. This information

can be used to create a new program containing only a subset of assertions and all their dependencies. We refer to this as "pruning" and the resulting program as a "pruned" program. Pruned programs provide an alternative visualization of the dependencies which resembles the structure of the original program. Moreover, since dependencies are defined as "everything required to prove the assertion", pruned programs should verify successfully. This property is exploited to implement sanity tests. In this section, we first describe in more detail how programs are pruned. Then, the implications of sanity test results are addressed.

Pruning a program with respect to a set of assertions Q proceeds as follows. First, all dependencies of Q are extracted through dependency analysis. Second, the original program is modified while preserving its structure. In particular, branches or loops are never removed in order to guarantee that the pruned program still verifies on all paths. Instead, if the branch or loop condition is not a dependency, it is replaced by their non-deterministic equivalent. Otherwise, the original condition remains unchanged. Further, declarations of methods, functions, domains, and variables are preserved. Irrelevant, primitive statements, such as assignments and method calls, invariants, and pre- and postconditions are removed.

An example is given in Listing 5.2 where the assignments on line 7 and 9 are identified as the dependencies of the assertion on line 11. They form the foundation of the pruned program provided in Listing 5.3. The difference to the original program is that the pruned program does not contain the precondition, assignment on line 5, and the branch condition. The latter is replaced by a non-deterministic branch denoted by `if(*){...}`.

```
1  method foo(a: Int)
2      requires a > 0
3  {
4      var n: Int
5      n := 0
6      if(a < 10){
7          n := 1
8      }else{
9          n := 2
10     }
11     assert n >= 0
12 }
```

Listing 5.2: The original Viper program before being pruned.

```
1  method foo(a: Int)
2  {
3      var n: Int
4
5      if(*){
6          n := 1
7      }else{
8          n := 2
9      }
10     assert n >= 0
11 }
```

Listing 5.3: The Viper program resulting from pruning the program on the left-hand side with respect to its assertion.

The result of executing a sanity test, i.e., verifying a pruned program, can be interpreted as follows. If the pruned program does *not* verify, this suggests that the analysis result for assertion set Q was unsound. This follows directly from our definition of dependency (Definition 3.6, page 14). On the other hand, successfully verifying a pruned program does *not* guarantee that the dependency set extracted from the analysis was sound, as demonstrated by the example in Listing 5.4. Here, an analysis mistakenly reporting the `assume` statement as a dependency instead of the assignment would pass the pruning test.

```
1  var a: Int
2  assume a > 0
3  a := 10
4  assert a > 0   // (Q)
```

Listing 5.4: An example demonstrating why pruning tests might report false positives, e.g., accept an unsound dependency result. The expected dependency is the assignment. However, an analysis reporting the explicit `assume` statement instead would be accepted by the automated pruning test.

In conclusion, verifying pruned programs can be used as a sanity test to increase confidence in dependency analysis results but it does not guarantee soundness of the results. Further, pruned programs provide an intuitive visualization of dependencies of assertions since the structure of the program is preserved.

Chapter 6

Improvements

The algorithm discussed so far detects dependencies in Viper but it is still imprecise in many cases. One major issue is the fact that we are merging *all* low-level nodes belonging to the same source. By doing so, a lot of detail is lost and the transitive closure is over-approximated. Second, some implementation details of Silicon are exposed to users, for example, state consolidation and the encoding of field assignments. We address how to resolve such issues in this chapter.

We start by introducing an alternative to merging nodes in order to preserve relevant low-level details and thus achieve better precision in Viper dependency graphs. Next, we discuss how to resolve imprecision issues present in the low-level dependency graph and how to hide implementation details. We finish by addressing improvement potentials for the alternative dependency definition introduced in Section 3.5 (Discussion and Refined Definitions).

6.1 Increasing Precision in the Viper Dependency Graph

One source of imprecision is the merging of *all* low-level nodes having identical source into one Viper node. This approach has two downsides. First, nodes store information relevant to users, for example, the assumption type. This information needs to be preserved to allow for fine-grained filtering in graph queries. Second, postconditions assumed as a result of a method call are merged into one node which hinders detection of *which* postcondition verification depends on. Figure 6.1 illustrates this issue by depicting a Viper dependency graph that imprecisely suggests that the client's assertion depends on both postconditions of `foo`.

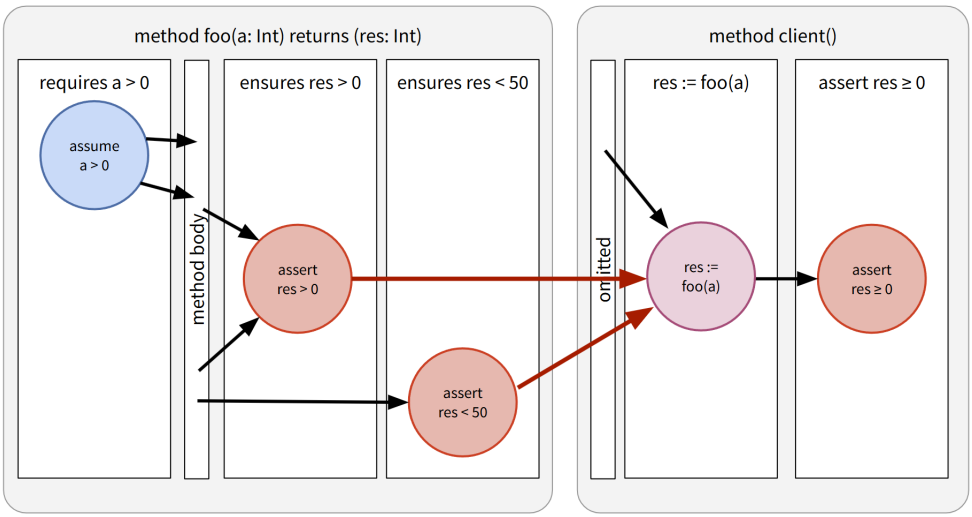

Figure 6.1: A Viper dependency graph illustrating how imprecision arises from the fact that the assumption of postconditions of method calls are merged into one Viper node. In reality, `client`'s assertion depends only on one of `foo`'s postconditions. The underlying Viper program is shown in Listing 4.6, page 42, and the low-level dependency graph in Figure 4.10, page 43.

Merging nodes enabled the detection of transitive dependencies. However, the same result can be achieved by adding edges between nodes of the same source in the low-level graph instead, as indicated by blue edges in Figure 6.2. As a result, transitive dependencies are detected on the lower level.

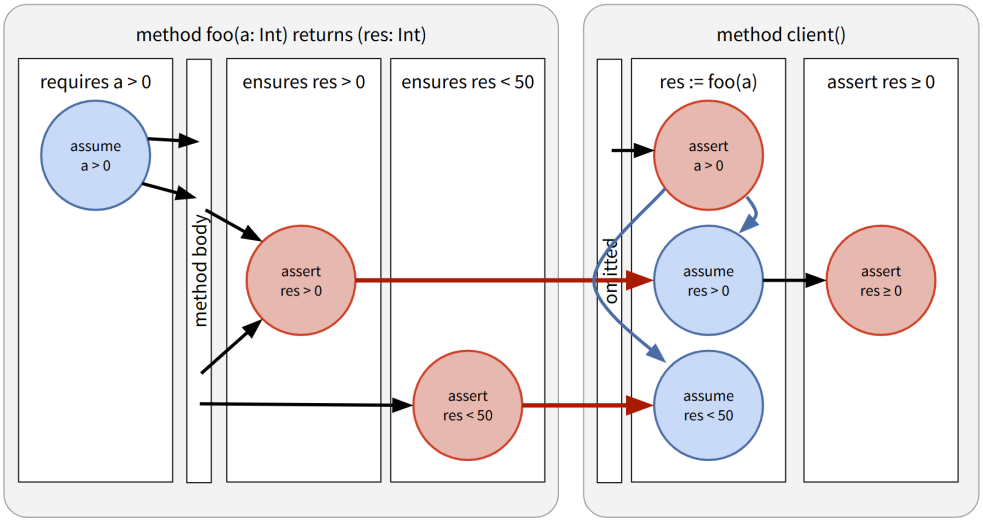

Figure 6.2: A low-level dependency graph with (blue) edges added between nodes having identical source, enabling the detection of transitive dependencies. The underlying Viper program is shown in Listing 4.6, page 42.

We can then compute dependency analysis queries on the resulting low-level graph and merge all nodes of identical source just before presenting the graph to the user. This allows precise computation of dependency sets using all low-level details but ensures that irrelevant details are hidden from users. Consequently, joining graphs via postconditions and method calls is more precise, as illustrated with the example in Figure 6.2, page 52. If a user queries for the dependency set of the client's assertion in this graph, the result contains only one postcondition of `foo`.

Note that our implementation slightly differs from the outlined approach. In particular, we merge low-level nodes that are identical in all information relevant for the precise computation of queries before adding edges. The motivation is to make the graph more compact, thus saving memory space and increasing the performance of graph queries.

In the remainder of this thesis, we refer to the process of adding edges and merging nodes as "lifting the graph to Viper." In visualizations, we usually omit edges between nodes of the same source and instead indicate this by using boxes, analogously to Chapter 4 (Identifying Dependencies in Silicon). If not stated otherwise, all edges between nodes of the same source are added as discussed above.

6.2 Increasing Precision in the Low-Level Dependency Graph

In the previous section, we discussed an optimized approach to lifting low-level graphs to Viper. In this section, we build on this by customizing the information stored in low-level nodes in order to increase precision of the low-level dependency graph. First, we discuss two common patterns that cause imprecision and propose for each one a technique used to customize node information. Depending on the implementation details of how the assumptions are derived, one or the other technique might be better suited. In the end, both techniques lead to the same result when applied properly. Then, we outline some situations, such as heap summary and state consolidation, where the proposed improvement techniques can be applied. Lastly, we address function axioms, which have to be split into smaller axioms to make analysis precise.

Grouping with Dedicated Sources

A major problem is that statements may introduce several, independent assumptions; for example, a statement might trigger a heap summary and thus all assumptions introduced by the heap summary are treated as assumptions made by that statement, i.e., the statement is used as a source. Consequently,

assertions depending on the heap summary inherit the statement's dependency and vice versa.

To separate independent assumptions, we propose to group certain assertion and assumption nodes by giving them a custom source. When lifting the graph to Viper, each group of assertions and assumptions is separated from everything else while edges within the group are added. Some nodes form their own group. We say that these are closed nodes and they can be marked accordingly. To guarantee soundness, it must be ensured that an assumption and all assertions required to justify it belong to the same group.

Assumption Labels

The previous technique is not applicable to assumptions composed of several independent parts since we cannot associate each part with a dedicated source. For example, state consolidation might introduce an assumption like $a \neq x \ \wedge a \neq y$. The goal is to track dependencies to both conjuncts independently such that an assertion depending on $a \neq x$ does not unexpectedly depend on inhaling permission to y which is a dependency of the second conjunct. One might think that we can simply split the assumption into its top-level conjuncts. While this is true in theory, doing so is not always possible due to implementation details of deriving such assumptions.

To deal with such cases, we propose a second technique where assumptions are conditionalized on so-called assumption labels, for example, $L1 \Rightarrow a \neq x$ where $L1$ is an assumption label. While deriving a composed assumption, each conjunct can be conditionalized on a dedicated label. The assumption label is a fresh, internal variable and is assumed to be true. A corresponding assumption label node is created when pushing it to the path condition. With this technique, an assertion depending on $a \neq x$ now also depends on $L1$. This enables fine-grained control because labels indicate the relevant conjunct. By carefully customizing the assumption nodes of the composed assumption and the labels, dependencies can be detected with high precision.

The intended use is depicted in Figure 6.3. In this example, state consolidation is triggered by the assertion and introduces a composed assumption establishing non-aliasing between several heap resources. The composed assumption (yellow node) is marked as closed such that it has no incoming edges. Each of its conjuncts, $x \neq y$ and $y \neq z$, is conditionalized on a dedicated label, $L1$ and $L2$, respectively. Each label reflects all dependencies required to justify its conjunct. Note that this is strictly necessary to guarantee soundness. Each label node has a dedicated source such that there are no edges between label nodes, as indicated by the dashed line. This is key to achieve precise results and was the main motivation to introduce label nodes in the first place.

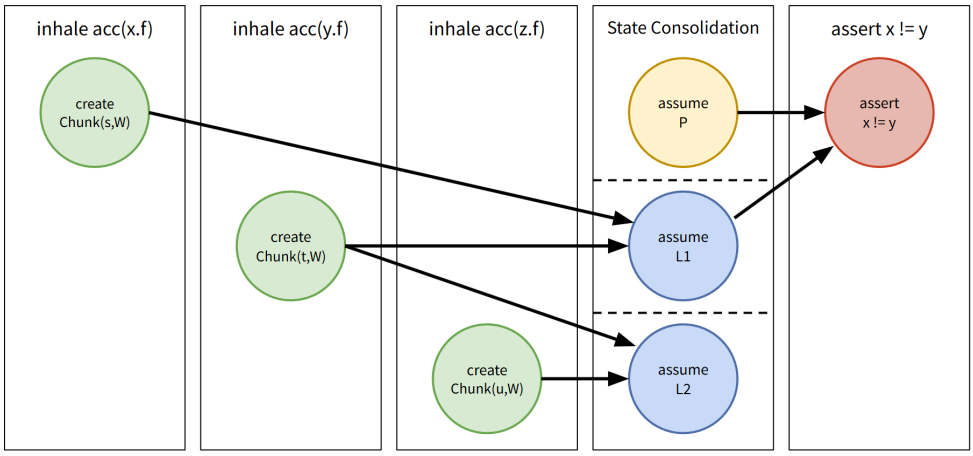

Figure 6.3: A demonstration of the intended use of assumption labels for a composed assumption $P := (L1 \Rightarrow x \neq y) \wedge (L2 \Rightarrow y \neq z)$. The nodes assuming $P, L1, L2$ are separated, i.e., there are no edges between them. Without the adjustments discussed in this section, these edges would have been added and the assertion would indirectly depend on all three `inhale` statements.

Heap Summary and State Consolidation

These two techniques are used, for example, to make heap summary and state consolidation more precise. We briefly addressed these internal operations, which are applied to make verification more complete, in Section 2.3 (Symbolic Heap Model). They summarize permissions, merge chunks, or assume non-aliasing of heap resources. In the following, we discuss how imprecision can be introduced when depending on assumptions derived from such internal operations as well as how to optimize them for more precision.

As heap summary can be triggered by any statement, its assumptions might get mixed with assumptions coming from the statement itself causing imprecision. We prevent this by giving nodes created during heap summary a dedicated source that is different from the statement's source. It ensures that no edges are added between these two groups unless there is a real dependency.

The same technique is applied to state consolidation. More concretely, a custom source is used when merging two chunks. We create one group per merged chunk containing the assertion proving aliasing as well as the create node of the merged chunk. This grouping is key to guarantee precision when several merged chunks result from a single state consolidation.

State consolidation has the additional issue that it introduces several unrelated assumptions in a composed formula, such as $x \neq y \wedge a \neq b$. As seen earlier in this section when introducing assumption labels, such composed assumptions cause imprecise analysis results. This issue is resolved by using

assumption labels and custom sources as explained above and visualized in Figure 6.3, page 55.

Conditional Permissions

Imprecision in permission requirement checks is further caused by heap summary. We demonstrate this on the following example. Assume we are given heap references `x, y` with permissions `pX, pY`, respectively, and `x != y` cannot be guaranteed. In this case, the heap summary has the following form

```
forall r:Ref. (r == x? pX : Z)+ (r == y? pY : Z).
```

It is consulted to determine whether read access to reference `x` is available. Apart from the expected dependency to the heap summary itself and permission to `x`, e.g., `pX == 1/2`, we observe that the SMT solver also reports permission to `y`, e.g., `pY == 1/4`. This is because the SMT solver needs to ensure `pY >= Z` to guarantee that the total permission to `r` is positive. However, permissions are always guaranteed to be non-negative and thus we do not expect this dependency to `pY`.

When instead using concrete permissions, read permission to `x` can be proven without depending on the permission amount for `y`; for example, for heap summary

```
forall r:Ref. (r == x? 1/2 : Z)+ (r == y? 1/4 : Z)
```

non-negativity of the second part holds trivially. Note that this is the heap summary as created when *not* using the simplified store model since concrete permissions are stored in chunks. Unfortunately, using concrete permissions would break soundness of the analysis as discussed in Section 4.2 (Collecting Assumptions) when introducing the simplified store model.

To ensure soundness and increase precision, we propose to conditionalize permissions on assumption labels before creating a chunk. In other words, chunks hold permissions in the form `L1? 1/2 : Z`, where `L1` is an assumption label as introduced earlier, such that no permission is granted when the assumption label cannot be proven. The advantage of this approach is that concrete permissions are used whenever possible. A fresh assumption label is created for each chunk and an edge from the chunk's create node to the assumption label's node is added explicitly.

This approach guarantees soundness of all permission requirement assertions because the assumption label prevents Silicon from simplifying permissions. In particular, the assumption label must be proven whenever permission to a chunk is required, which invokes the SMT solver, as is the case with the simplified store model. Through the UNSAT core, the analysis correctly detects the dependency to the label node and thus, indirectly to the create node, as visualized in Figure 6.4.

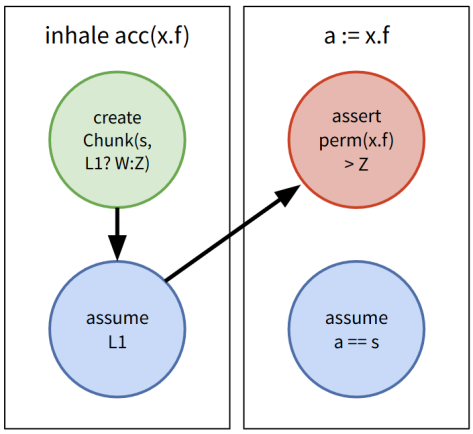

Figure 6.4: A visualization of how permission to `x.f` gets conditionalized on an assumption label `L1`. Note that `L1? W : Z` is stored in the chunk but not represented in the path condition or by an assumption node. Instead, `L1` is added to the path condition. When asserting permission requirements, Silicon retrieves this permission, creates the SMT solver query `(L1? W : Z) == W`, and a dependency to `L1` is reported.

Further, it helps increasing precision for the heap summary which now has the form of

```
forall r:Ref. ((r == x && L1)? 1/2 : Z)+ ((r == y && L2)? 1/4 : Z).
```

Here, the only assumption needed to prove read access to `x` is `L1` because the second part, `((r == y && L2)? 1/4 : Z)`, is trivially non-negative, independent on whether `x` and `y` are aliases. Importantly, no dependency to `y` or `L2` is reported making the analysis precise.

Note that not all chunks hold concrete permissions. For wildcards and permission variables, e.g., `acc(x.f, p)`, a fresh, internal variable representing the permission amount is created and stored in the corresponding chunk. Thus, heap summary, and potentially other operations, might still lead to imprecision in these cases. We leave this for future work.

In the remainder of this thesis, we omit these details in visualizations unless relevant. Importantly, a create node with `Chunk(s, 1/2)` is an abstraction of two nodes, namely the create node `Chunk(s, L1? 1/2 : Z)` and assumption label node `L1`.

As a side note, an alternative might have been to use the simplified store model on permissions and additionally add internal assumptions of the form `p >= Z` for all permission variables. The corresponding assumption node should not have any dependencies such that any assertion can depend on it without introducing imprecision. However, it is not inherently clear how to ensure that the UNSAT core reports this assumption instead of the assumption establishing the actual permission amount, e.g., `pY = 1/4`.

Field Assignments

Another implementation detail concerns field assignments. From a Viper user's perspective, assignments do not remove or add permission to any heap location and, thus, should not be reported as a dependency to subsequent permission requirement assertions. However, field assignments are implemented as removing the existing chunk and creating a new chunk. As a consequence, any subsequent access to the assigned resource depends on the assignment since this statement created the corresponding chunk. More importantly, this create node depends on every assertion made as part of verifying the assignment. An example where this causes imprecision is presented in Listing 6.1 where the `exhale` statement unexpectedly depends on both preconditions.

```
1 method foo(x: Ref, a: Int)
2   requires acc(x.f)
3   requires a > 0    // not a dependency!
4 {
5    x.f := 10/a
6    exhale acc(x.f)  // test assertion
7 }
```

Listing 6.1: An example where a field assignment causes imprecision if not handled carefully. In particular, dependency analysis would report a dependency from the chunk created at assignment in line 5 to the `exhale` statement in line 6. Further, the precondition `a > 0` would be inherited as a dependency over line 5 since this is needed for the division.

This imprecision can be resolved by splitting the assignment into the left-hand and right-hand side and associating the corresponding low-level nodes with different sources. Importantly, the left-hand side contains the remove and create node of the assigned heap resource. Due to having different sources, the create node does not depend on the right-hand side anymore which resolves the imprecision. On the other hand, the equality resulting from the assignment, e.g., `x.f == 10/a`, is part of the right-hand side and should depend on all assertions made for the assignment, namely also asserting full permission to the left-hand side. Hence, we explicitly add edges pointing from the left-hand to the right-hand side, but not vice versa. The resulting dependencies are visualized in Figure 6.5.

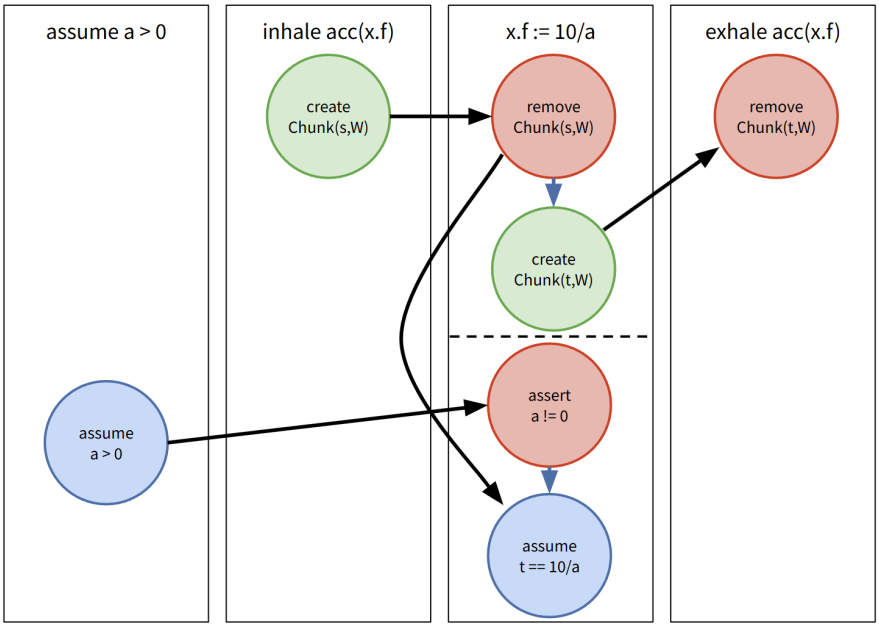

Figure 6.5: A visualization of how field assignments are handled by the dependency analysis. The dashed line represents the splitting into two sides. For clarity, blue edges indicate all edges added when lifting the graph to Viper. Note that the create node of the assignment does not depend on assertions of the right-hand side of the assignment (below the dashed line). As a consequence, `assume a > 0` is not reported as a dependency of the `exhale` statement. On the other hand, if an assertion depends on the assigned value `10/a`, it would indirectly depend on the `inhale` statement *and* on `assume a > 0`.

Function Axioms

So far, we discussed imprecision caused by the symbolic heap and its implementation details. However, functions, pure and impure, also cause imprecision. The underlying cause is that Silicon creates only one function axiom representing all postconditions and the function body at the same time. It has the form

```
forall args. precondition => (postconditions && result == body).
```

While the unoptimized analysis is able to detect whether an assertion depends on this axiom or not, it cannot determine on *which* postconditions or body it depends.

In order to detect dependencies in a more fine-grained manner, we instead create one axiom for each postcondition

```
forall args. precondition => postcondition
```

and another one for the function body

```
forall args. precondition => (result == body).
```

This is semantically equivalent to having one axiom covering all of them but now the analysis can detect dependencies to each postcondition and the body independently.

An alternative idea would be to use assumption labels similar to the improvement for state consolidation. However, introducing assumption labels always comes with a small overhead since more nodes are created. Splitting axioms achieves the same outcome in a more elegant fashion.

6.3 Hiding Silicon Implementation Details

The previous section discussed how to increase precision in various cases including internal operations such as state consolidation and heap summary. However, artifacts of these improvements, e.g., assumption label nodes, and the underlying internal operations should be hidden from the user. To this end, nodes can be labeled as internal using the corresponding assumption type introduced in Section 5.1 (Assumption and Assertion Types). It is important to note that internal assumptions are never exposed to the user, but transitive closure is preserved. In other words, a dependency chain $A \rightsquigarrow B \rightsquigarrow C$, where only B is internal, is presented to the user as $A \rightsquigarrow C$.

During verification, several assumptions are marked as internal, most importantly, every assumption made by state consolidation and heap summary as well as create nodes created by field assignments. All of those are derivations of existing assumptions and implementation details, as already discussed in the previous section. Additionally, we hide implementation details of the dependency analysis by marking assumption label and infeasibility nodes as internal.

Further, assumptions introduced by explicit `assert` and `exhale` statements are marked as internal. In Silicon, these statements add the asserted fact to the path condition after proving it, thus adding an assumption node. As a consequence, assertions can depend on preceding assertions. From Viper's perspective, assertions do not add any new facts and should therefore not be treated as assumptions. The dependencies of the assertions should be reported instead. This solution is visualized in Figure 6.6 where gray nodes indicate internal assumptions. Thus, only the assignment is reported as a dependency for both assertions.

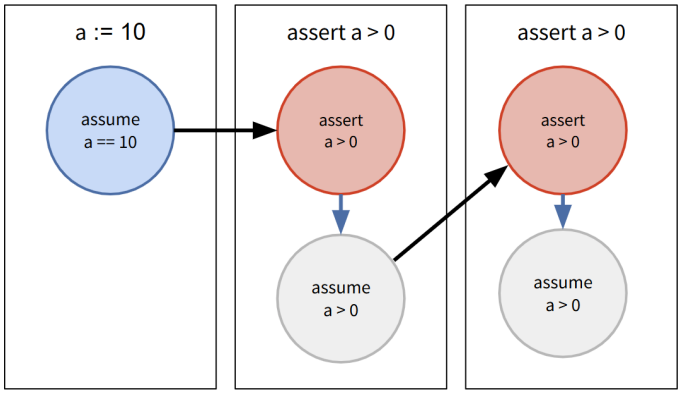

Figure 6.6: A demonstration of how implementation details of `assert` statements are hidden. Gray nodes indicate internal assumptions which are hidden from users.

6.4 Improvements towards the Alternative Definition

Lastly, we propose improvements that could be explored in future work. We did not implement them in this thesis because they break soundness of the dependency analysis with respect to our definition, given in Section 3.3 (Dependency). However, these improvements are sound and make the analysis more precise for the alternative definition addressed in Section 3.5 (Discussion and Refined Definitions). This alternative definition is motivated by use cases reasoning about dependencies in *concrete* executions.

We briefly present three possible improvements and how they could be implemented. We deem it possible to add a configuration option that controls which definition should be used. Potentially, they can even be implemented as postprocessing operations on the final dependency graph such that the same underlying dependency graph can be used to execute queries for both definitions.

Path-Awareness

As seen in Section 4.5 (Branches and Loops), the low-level dependency graph is path-aware in the sense that edges are only added between nodes discovered on the same path. However, this property is lost when lifting the graph to Viper. To preserve path-awareness, the nodes would need to store information about the current path. The operation lifting the low-level graph to Viper needs to be adjusted to keep nodes of disjoint paths separate. A challenge might be to correctly handle loop invariants to guarantee that dependencies between the loop body and statements outside the loop can be detected.

Joining Methods via Preconditions

According to our definitions, an assertion depending on one postcondition of a method call inherits the dependencies of asserting *all* preconditions. This is even the case if some preconditions are irrelevant for proving the required postcondition during verification of the callee. We could change this behavior by not adding any edges from pre- to postconditions of the method call and instead joining graphs via postconditions *and* preconditions. In other words, the connection between pre- and postconditions of a method call is made over the callee's graph.

Predicates

Similarly to methods, we argue that an assertion relying on an `unfold` statement does not necessarily depend on everything needed for the corresponding fold operation. This is because the assertion might require only part of the predicate's body and hence should only depend on assumptions establishing that part. Optimizing the algorithm to detect these fine-grained dependencies is more involved than handling method calls. However, it might be possible to implement graph operations to find nodes created by the fold operation that preceded the unfold. Removing edges between a fold and its subsequent unfold, and instead adding edges between matching parts of the predicate body might be an option, but exploring this idea is left for future work.

Chapter 7

Implementation

The previous chapters thoroughly explained and discussed the dependency analysis algorithm. In this chapter, we address how we implemented this algorithm in Silicon.

The high-level architecture is visualized in Figure 7.1. Green nodes represent existing Silicon components. Purple nodes are used for components relevant for constructing the low-level dependency graph during verification. Finally, blue nodes represent components related to presenting and interpreting the results after verification terminates. The edges represent the main interactions and data flow. The edge going through the dependency analyzer indicates that this component translates the low-level dependency graph to a Viper dependency graph. The components implementing the main functionality, namely the dependency analyzer and graph interpreter, are emphasized. Moreover, data structures with no or little logic are visualized with dotted outlines. Note that the analysis is not enabled by default and can be enabled by setting the `--enableDependencyAnalysis --disableInfeasibilityChecks` configuration flags.

We approach this chapter by first describing the main modifications made to various Silicon components. Subsequent sections briefly describe each analysis component and their interaction with Silicon components. Later, we address how dependency analysis is lifted to Gobra [33], a Viper frontend for verifying Go programs. We finish by discussing the implementation of annotated tests used for the evaluation in the next chapter.

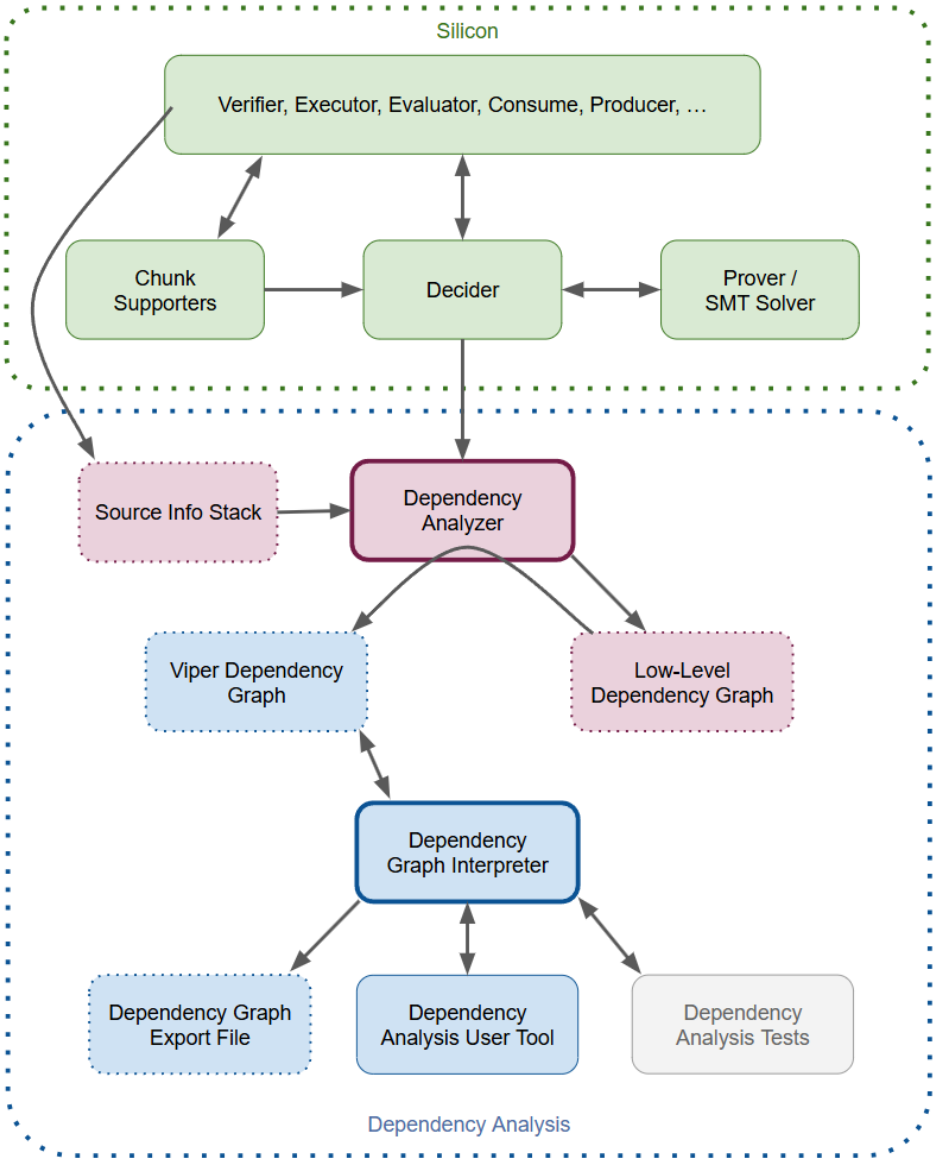

Figure 7.1: The high-level architecture of the dependency analysis implemented for Silicon.

7.1 Modifications to Silicon Components

To enable dependency analysis, we had to slightly modify some Silicon components. We discuss the main modifications in this section.

Simplified Store Model

Enforcing the simplified store model, proposed in Section 4.2 (Collecting Assumptions), is rather straightforward because Silicon already uses it when the assigned symbolical value is a composed term, e.g., $x + 1$. Variable and literals, on the other hand, are usually directly set in the store's mapping. We disable this distinction such that Silicon treats variables and literals as composed terms, thus, creating a fresh, internal variable and updating the store's mapping accordingly.

Prover

The prover interacts with the SMT solver by adding assumptions and querying for assertions. We adapted the SMT solver calls such that (1) all assumptions are labeled and (2) all successful assertions extract the UNSAT core. The former is implemented using Z3's `named` attribute [18] and the label is received as an input from the decider. For (2), we invoke the `(get-unsat-core)` query which returns the UNSAT core as a set of labels. The UNSAT core is given to the decider.

Decider

The decider is a central data structure holding a reference to the prover, the path condition stack, the dependency analyzer, and the source information stack, among other things.

It implements assume and assert functions which invoke the corresponding functions provided by the prover. We modified these functions such that they take an assumption type as input and interact with the dependency analyzer by passing the assumption or assertion as well as the assumption type and source information. The source information is retrieved from the source information stack which provides all information necessary for the dependency analyzer to create nodes accordingly. After creating a node, the dependency analyzer returns the label of that node which is sent to the prover and used as a label in SMT solver queries. For successful assertions, the decider retrieves the UNSAT core from the prover and sends it to the dependency analyzer which adds dependencies accordingly.

Various Silicon Components

The decider's assume and assert functions are called by various Silicon components, for example, by the executor, evaluator, consumer, and producer. We adapted many functions in these components to pass around assumption types which are required as an input for the assume and assert functions as well as chunk constructors. The assumption type is usually determined by the statement that is currently being executed and is passed down to

other components. Further, Silicon components maintain the source info stack by pushing and popping the currently executed statement or evaluated expression.

Chunk Supporters

In order to keep track of all chunks, we adapted the chunk constructors such that they register all newly created chunks to the decider. The decider passes this information to the dependency analyzer which adds a create node accordingly. Similarly, all chunk methods, such as `permAdd` or `permMinus`, register newly created chunks and deregister removed chunks. Registering chunks also wraps the permission with an assumption label and adds the label to the path condition. Creation of the corresponding label node is handled by the dependency analyzer.

Exploring Infeasible Paths

As discussed in Section 4.6 (Dependencies to Control Flow Conditions), infeasible paths need to be executed to add dependencies to the proof of infeasibility. Since infeasibility is detected by asserting `false`, the decider is involved in this process and passes this information to the dependency analyzer which creates an infeasibility node and returns the node's label. This label is then stored in the current path condition stack layer as a proof of infeasibility, which guarantees correct propagation of the label, for example, when returning to a previous state in order to explore a different path.

Usually, verification terminates successfully after detecting infeasibility. We overrule this behavior by continuing verification as if infeasibility was not detected. Additionally, the decider skips all assumptions and assertions whenever an infeasibility label is set in any path condition layer. Instead, for assertions, the dependency analyzer creates an assertion node with a dependency to the infeasibility label, i.e., the infeasibility node. These modifications ensure that all infeasible paths are executed until the end and all encountered assertions are represented by a node with a dependency to the infeasibility proof.

State Consolidation and Heap Summary

As discussed in Section 2.3 (Symbolic Heap Model), Silicon triggers state consolidation and heap summary to make verification more complete. Since these are implementation details, all assumptions introduced by these operations are marked as internal. Further, they get a customized source such that these internal nodes are not mixed with nodes created by the statement that triggered state consolidation or heap summary. To this end, the source

information stack is overwritten at the beginning of these operations and restored at the end.

Domain and Function Axioms

Domain and function axioms are a special case because they are added directly to the prover, circumventing the decider. In order to track them, we had to make two major adjustments.

First, the prover needs access to a dependency analyzer in order to create the corresponding assumptions nodes. Such a dependency analyzer is created at initialization of the prover and stored directly in the prover. Second, when a dependency analyzer is initialized for a specific program component, the prover already knows the axioms and has associated a label with it. In order to track dependencies to those, the dependency analyzer needs to create nodes accordingly. This is done by accessing the prover's dependency analyzer and extracting its nodes. Note that this is necessary to ensure that the axiom labels used by the SMT solver are identical to the node identifiers, i.e., creating new nodes with new identifiers would not solve the problem.

7.2 Dependency Analysis Components

To detect and keep track of dependencies, we designed new Silicon components, most importantly, the dependency graph data structure and the assumption analyzer. They are discussed in this section.

Dependency Graph

The underlying data structure to represent dependency graphs is a simple graph structure storing nodes and edges. Graph nodes are stored in a simple list. There are different types of nodes such as assume, assert, create, and infeasibility nodes. They differ in the information stored in them, for example, create nodes store the associated chunk while assume and assert node store an expression instead. Edges are maintained as a map of node identifiers to a set of node identifiers. These identifiers uniquely identify the node and are identical to the labels used in SMT solver queries. Due to this edge representation, adding dependencies extracted from the UNSAT core amounts to extracting the labels and adding them directly to the map without having to access any node. Various functions for adding nodes and edges are exposed which are mainly used by the dependency analyzer.

Note that we use the same data structure to store low-level and Viper dependency graphs. This has the advantage that the low-level graph can be lifted easily to a Viper graph. Furthermore, the same interpreter implementation

can be used to operate on both graphs. This is exploited for debugging purposes where analysis queries are executed directly on the low-level graph.

Further, the dependency graph provides functions to compute dependency and dependents sets of given assertions or assumptions, respectively. The algorithm for computing dependency sets employs a queue-based approach. The queue is initialized to the set of assertions. Then, an element of the queue is visited by adding all its direct dependencies, given by the edges, to the queue and the result set. This step is repeated until the queue is empty. The algorithm for computing all dependents of given assumptions is analogous.

Lastly, we define a read-only interface to the dependency graph class which is used by the interpreter. It only includes access to nodes and edges as well as the dependency and dependents query. The reason for this is to decouple graph construction from graph interpretation and querying such that the graph cannot be modified during interpretation.

Dependency Analyzer

The central, most important data structure for construction of the dependency graph is the dependency analyzer. It has two implementations: A default implementation used when assumption analysis is enabled, which is described shortly. Otherwise, a no-op implementation is used that does absolutely nothing, i.e., it does not create any nodes or edges, and has no impact on verification. When initializing the dependency analyzer, the implementation is chosen depending on the configuration flag indicating whether analysis is enabled. Having a no-op dependency analyzer simplifies implementation because dependency analyzer methods can be invoked without having to repeatedly check the configuration flag.

The dependency analyzer is initialized once per program component and creates one dependency graph. When verification of this component terminates, the graph is finalized and attached to the verification result. Graphs of all program components can be joined once verification of all of them terminates. The join operation uses assumption types and source information to identify nodes of different graphs corresponding to the same postcondition and adds edges accordingly.

As shown in Figure 7.1, page 64, the dependency analyzer acts as a mediator between the decider and the dependency graph. Most importantly, it creates nodes based on the information provided by the decider and adds them to the graph. Further, it parses UNSAT cores to determine which edges to add.

Moreover, the dependency analyzer exposes the necessary functions such that Silicon components can add custom edges or mark nodes as closed. Further, it is responsible for creating assumption labels and conditionalizing expres-

sions, such as permission, on them. These functions enable improvement techniques, mainly for increasing precision.

Lastly, the dependency analyzer implements the logic required to lift a low-level dependency graph to a Viper dependency graph by adding edges between nodes as discussed in Section 6.1 (Increasing Precision in the Viper Dependency Graph). While finalizing the graph, it also removes all label nodes and merges identical nodes, i.e., nodes with the same source and assumption type. This is a minor optimization to make the graph more compact and not unnecessarily complicate graph queries. These operations preserve the transitive closures of dependencies to guarantee soundness.

Source Information Stack

The most crucial information required to lift low-level graphs to Viper graphs is the source. It is maintained in the source information stack which is attached to the decider and thus available to all relevant components.

Whenever the execution of a statement starts, the statement is pushed to this stack. Similarly, expressions are pushed when starting evaluation. Once the corresponding operation returns, the element is popped. In other words, some Silicon methods are wrapped by calls to push and pop such that the source is tracked correctly. Note that the statements and expressions are associated with their position in the source code and, consequently, this information is also stored in the nodes. This is important to make statements and expressions uniquely identifiable when lifting the low-level graph and when users are analyzing the dependency graph.

Branches had to be handled carefully because in that case, a statement's execution might return several times. This was resolved by explicitly setting the source information stack at the beginning of each branch. Furthermore, sometimes the source information stack is overwritten and restored later on. This technique is used, for example, to hide state consolidation and heap summary as discussed in Section 6.3 (Hiding Silicon Implementation Details).

The decider can extract the current source from this stack and pass this information to the assumption analyzer. The source always corresponds to the bottom of the stack, i.e., the statement added first. Nodes also store a so-called fine-grained source which corresponds to the top of the source, i.e., the statement added last. This is used to join graphs of different methods because for method calls the fine-grained source conveniently corresponds to the assumed postcondition.

7.3 Querying the Graph

We provide two options for querying dependency graphs. First, the graph can be queried through a command-line tool that operates directly on the dependency graph. This tool is based on the dependency graph interpreter, which is discussed first. Second, we provide the option to export the dependency graph to a CSV format such that it can be easily imported to existing graph analysis tools.

Dependency Graph Interpreter

As the name suggests, the dependency graph interpreter is given a read-only view of a dependency graph and interprets it. That is, it provides various graph operations which can be used for analyzing the graph and extracting useful information from it. It is initialized once verification of all program components terminates and the final dependency graph is available. It is the foundation for the command-line tool as well as automated tests and benchmarks.

The interpreter provides convenience functions to extract dependency nodes with certain properties, for example, all explicit assumptions or all assertions belonging to a certain source code line. They are implemented by retrieving all nodes from the dependency graph and postprocessing them accordingly, e.g., by filtering or grouping nodes.

To compute dependency or dependents sets of given assertions, the interpreter invokes the corresponding functions provided by the dependency graph itself and applies filters to the result set if necessary. It is crucial to apply filters *after* computing the dependency set to ensure that all transitive dependencies are reported. In particular, for dependency set queries, all assertion and internal assumption nodes are filtered out by default.

The interpreter further computes proof coverage as defined in Section 5.2 (Proof Coverage). The implementation is straightforward. It involves computing the according dependency set, extracting all nodes from the graph, and filtering both sets for the program component.

Moreover, the interpreter provides the functionality to export graphs to CSV files such that graphs can be analyzed in other tools. To this end, a CSV file containing all nodes and their properties as well as a CSV file containing all edges is created. Together, these files encode all information necessary to reconstruct the dependency graph. Users can set the configuration flag `--assumptionAnalysisExportPath [PATH]` such that the graph is automatically exported after verification terminates.

Lastly, the interpreter contains the functionality to prune Viper programs. The implementation iterates over the original program and replaces all irrelevant

nodes as discussed in Section 5.3 (Pruning Viper Programs).

Dependency Analysis User Tool

As a first analysis option, we designed a command-line tool for analyzing dependency graphs directly inside the Viper tool chain, without the need to set up other tools and import graphs into those. This tool starts automatically after verification of the program terminates if the configuration flag `startAssumptionAnalysisTool` is set and interacts with the user via the command-line. All received user queries are delegated to the dependency graph interpreter and the returned results are presented to the user. This tool supports the following queries:

1. Print all dependencies of a given set of assertions.
2. Print all dependents of a given set of assumptions.
3. Determine whether there exists any dependency between any pair of nodes in a given set of nodes.
4. Compute the proof coverage of a given set of assertions.
5. Compute the proof coverage of a given method.
6. Prune the program with respect to a given set of assertions.

Queries 1 and 2 return four result sets each in order to give users the option to choose which information to look at and potentially combine information from different sets. An example is depicted in Figure 7.2. The following result sets are provided:

(a) The first result set contains all dependencies except that it filters out internal assumptions to hide implementation details. All other dependencies, i.e., the transitive closure, is preserved. More formally, if a non-internal assumption A is a dependency of assertion Q, A is reported as a dependency, even if Q depends on A indirectly via an internal node.

(b) The second result set is analogous but it only shows nodes with the assumption type “explicit”. Of course, the transitive closure is preserved.

(c) Another result set is dedicated to direct dependencies.

(d) Slightly different to the above sets is the last result set which excludes dependencies introduced over infeasibility nodes and does *not* preserve the transitive closure. In other words, it returns the results as if infeasible paths were not executed. This is an experimental feature that might be useful to detect imprecise results caused by the execution of infeasible paths.

```
1  method foo(x: Ref)
2    requires acc(x.f) &&
         x.f > 0
3  {
4    var a: Int, res: Int
5    if(a > 0){
6      res := 10
7    }
8
9    res := x.f + 1
10
11   if(a > 0){
12     res := res + a
13     assert res > 0
14   }
15 }
```

```
1  Queried: res > 0 (line 13)
2
3  (a) all:
4    acc(x.f) (line 2)
5    x.f > 0 (line 2)
6    a > 0 (line 5)
7    res := x.f + 1 (line 9)
8    a > 0 (line 11)
9    res := res+a (line 12)
10
11 (b) explicit
12   acc(x.f) (line 2)
13   x.f > 0 (line 2)
14
15 (c) direct
16   x.f > 0 (line 2)
17   a > 0 (line 5)
18   res := x.f + 1 (line 9)
19   a > 0 (line 11)
20   res := res + a (line 12)
21
22 (d) without infeasibility
23   acc(x.f) (line 2)
24   x.f > 0 (line 2)
25   a > 0 (line 5)
26   res := x.f + 1 (line 9)
27   res := res + a (line 12)
```

Figure 7.2: An illustration of the command-line tool. The right-hand side shows the result when being queried for the dependency set of the assertion on the left. Note that query (d) does not report the branch condition of line 11 as a dependency since this dependency is introduced through an infeasible path.

In future work, e.g., when designing a visual user interface, result sets could be implemented as configuration options, for example, to filter over more fine-grained assumption types, e.g., invariants. Moreover, labeling a node as irrelevant and re-computing the dependencies with this additional information might be useful to filter out noise and thus get more precise analysis result.

Query 3 takes a set of assumptions and assertions and returns true or false depending on whether there is any dependency between any pair of given nodes. This is useful if one wants to quickly see whether statement a depends on statement b without getting overloaded with additional information.

Queries 4 to 5 return the proof coverage corresponding to the definitions addressed in Section 5.2 (Proof Coverage). These queries also return all

uncovered nodes.

Query 6 exports the pruned program as discussed in Section 5.3 (Pruning Viper Programs) to a file, enabling users to verify it to gain confidence in the analysis result, for example.

Neo4j Graph Import

The need to efficiently and intuitively analyze graphs is not a recent development and hence, many analysis tools have been developed already. One such tool is Neo4j [29], a database that natively stores graph data structures and enables querying graphs using the query language Cypher [1]. It is thus perfectly suited for our purposes. We implemented a script that takes graph export files, created by the interpreter, and imports the graph into a Neo4j database. It automatically creates two different views of the same graph. The first view contains all nodes including internal assumptions. It should be used for all graph queries since it allows precise computation of dependencies. In the second view, internal nodes are already hidden and all nodes belonging to the same source are merged. This graph is compact and user-readable, thus providing a good overview of all dependencies in the program. However, computing queries on it can lead to imprecise results because it misses important low-level information.

7.4 Discovering Dependencies in Gobra

So far, we extracted dependencies present in Viper programs. The dependency analysis can be lifted to yet another level, namely to frontends like Gobra. In the following, we assume we are given a low-level dependency graph and want to construct the corresponding Gobra dependency graph.

To this end, we need a mapping from low-level nodes to Gobra nodes. Conveniently, this has already be done in prior work, namely to enable error reporting. More concretely, nodes in a Viper program that has been generated from Gobra are already associated with their Gobra source, as opposed to the Viper source. Consequently, merging nodes as described in Section 4.1 (Assembling the Viper Dependency Graph) results in Gobra nodes. Note that the Viper dependency graph is never assembled. Instead, we lift the low-level graph directly to the corresponding Gobra dependency graph.

However, the graph contains some nodes subject to internal encoding of Gobra features in Viper. For example, string and array domains and associated utility functions are created in order to encode Go strings and arrays. Such nodes should be hidden from the user. We observe that nodes coming from encodings are not associated with a Gobra position since they are not part of

the Gobra program. This information is used to single out these nodes and mark them as internal to hide them from users.

The last step is to lift the assumption and assertion types to Gobra. This is necessary since, for example, a string initialization is internally encoded as inhaling permission to the string object. From a Gobra user's perspective, this is an implicit assumption. From Viper's perspective the `inhale` statement is an explicit assumption. To overcome this, Viper statements and expressions can be annotated with assumption and assertion types. This overwrites the default type assigned by Viper. For example, `@assumptionType("Implicit") inhale a > 0` introduces an implicit assumption node. These annotations are added to the Viper program during translation from Gobra to Viper. The assumption type is determined based on the Gobra node, e.g., a string initialization is implicit and a precondition is explicit.

In conclusion, lifting the Viper dependency analysis heavily relies on Gobra positions which are used to merge nodes of the low-level graph. Additionally, we add assumption type annotations. Note that Gobra support is experimental at the moment and this implementation for annotating assumption types is based on heuristics. Moreover, we tested the dependency analysis only on a fraction of Go language features. Extending it is left for future work.

7.5 Annotated Tests

To gain confidence in our implementation of dependency analysis, we designed a test framework that takes annotated Viper programs and checks whether the detected dependencies correspond to the annotations.

To this end, users may define one test assertion per method by annotating a statement or expression with `@testAssertion()`. Further, arbitrarily many Viper statements and expressions can be annotated with `@dependency()` or `@irrelevant()` depending on whether or not they are expected to be a dependency of the test assertion. The test framework checks whether all expected dependencies are detected, thus indicating soundness, and whether irrelevant dependencies are wrongly reported, thus checking precision.

The tests also verify that each annotated statement or expression is captured by at least one node in the graph. Moreover, the annotations can be enhanced by an assumption or assertion type, e.g., `@dependency("Explicit")`. In this case, the tests verify that a node with the annotated assumption type exists and that a dependency to such a node does or does not exist, respectively.

Chapter 8

Evaluation and Discussion

So far, we discussed the analysis algorithm and its concrete implementation in Viper's symbolic execution backend. In this section, we evaluate how it performs with respect to various metrics. We start by stating the research questions and proceed by answering each question in a dedicated section.

We base our evaluation on the following research questions which exercise different properties of the dependency analysis.

RQ1: Is the dependency analysis sound with respect to Definition 3.10, page 17?

RQ2: Is the dependency analysis precise with respect to Definition 3.11, page 17?

RQ3: How does dependency analysis affect the verification time of real-world programs? How do dependency graph queries perform?

RQ4: How accurately is proof coverage computed?

RQ5: Can the analysis provide insights that facilitate debugging processes in program verification, in particular concerning partially verified programs?

8.1 RQ1: Soundness

The most important metric of the dependency analysis is soundness which states that all existing dependencies are reported (see Definition 3.10, page 17). In the following, we present an informal soundness argument in the context of Viper and then describe how we tested our implementation with respect to soundness. Lastly, we briefly address soundness of the Gobra dependency analysis.

Soundness of the Viper Dependency Analysis

As an informal soundness argument, we need to show that the Viper dependency graph represents all assumptions and assertions as nodes and all dependencies between them as edges. We start by reasoning about the low-level dependency graph constructed using the optimized algorithm where dependency sets can already be computed on the lower level, as explained in Section 6.1 (Increasing Precision in the Viper Dependency Graph). To this end, we assume that Silicon and the SMT solver are sound.

First, the low-level dependency graph must contain all necessary nodes, which is stated by following two claims.

Claim 1: All assumptions are represented by a low-level assumption node.
The correctness of Silicon and the simplified store model guarantee that all assumptions are either pushed to the path condition or tracked as a chunk in the symbolic heap. Both cases are intercepted by the dependency analysis which creates a corresponding low-level assumption node.

Claim 2: All assertions are represented by a low-level assertion node.
It is guaranteed that all assertions are reached since we execute all paths, whether feasible or not. The simplified store model ensures that all assertions are verified by the SMT solver since all constraints on variables and values are encoded in the path condition and SMT solver, i.e., Silicon itself cannot prove any assertion. This includes assertions related to permission requirements since permissions are conditionalized on assumption labels which must be proven to hold by the SMT solver.

Next, we argue that all required edges are added to the low-level dependency graph. Claim 3 and 4 cover direct dependencies of assertions and assumptions, respectively.

Claim 3: All direct dependencies from assumptions to assertions are represented by corresponding edges in the low-level dependency graph.
Since all assertions are verified by the SMT solver, this follows by the correctness of UNSAT cores and the fact that the UNSAT core can be mapped to the corresponding assumption nodes. The latter holds since all assumptions are tracked (Claim 1).

Claim 4: All dependencies between assumptions and assertions directly required to justify them are represented by corresponding edges.
Such derived assumptions and their assertions are guaranteed to be made within the same operation. For non-optimized operations, these nodes have the same source and hence all necessary edges are added. Improvements, discussed in Chapter 6 (Improvements), explicitly require this claim to hold.

Exempted from this rule are composed assumptions where dependencies of each conjunct are encoded through assumption labels. Lastly, assumptions taken from other program components need to be taken into account, for example, postconditions of method calls. These cases are covered by explicitly joining graphs as well as collecting function and domain axioms, as discussed in Section 4.7 (Dependencies across Program Components).

Finally, we reason about soundness of low-level dependency graphs, as stated in claim 5.

Claim 5: All dependencies between low-level assumptions and assertions are represented in the low-level dependency graph.
The definition of soundness (Definition 3.10, page 17) requires to prove correctness of the computed dependency set (Definition 3.6, page 14) for each assertion. More precisely, for each assertion, it needs to be ensures that reported dependency set contains all direct dependencies, all transitive dependencies, and dependencies across program components. Dependency sets are computed by taking the transitive closure of dependencies. Claim 3 and 4 guarantee that these sets contain all direct dependencies and by induction also all transitive dependencies. Lastly, claim 4 already established dependencies across program components, which concludes this claim.

It remains to show that low-level dependency graphs are correctly lifted to Viper dependency graphs. For this purpose, claim 6 covers the merging of nodes and claim 7 reasons about dependencies represented in Viper dependency graphs.

Claim 6: Low-level nodes are correctly mapped to Viper nodes.
All low-level nodes having identical source are merged into one Viper node, thus the correctness of each node's source is crucial. Dependency analysis tracks the currently executed Viper node which is used as the source of each low-level node created while executing it. Custom sources are only assigned to internal assumptions and assertions, which are hidden anyway and thus irrelevant for this claim.

Claim 7: All Viper dependencies are represented by the Viper dependency graph.
Claim 5 already states that the low-level dependency graph is sound, i.e., it correctly represents all dependencies. Then, claim 7 follows by claim 6 and the fact that all edges are preserved when lifting the low-level graph to a Viper graph.

In conclusion, claim 7 argues that the Viper dependency graph is correctly constructed, meaning that it represents all Viper dependencies. The algorithm

heavily depends on the correctness of Silicon, the SMT solver, and the sources assigned to low-level nodes.

Automated Test Suite

To increase confidence in the correctness of our implementation, we designed a test framework and various unit tests, which can be found in the Silicon GitHub repository [6]. The tests cover a large fraction of Viper features including, among others, loops, branches, predicates, magic wands, and quantified permissions. Each unit test is a small Viper program exercising a specific feature. The framework is based on annotated tests described in Section 7.5 (Annotated Tests). For each method in these programs, we defined one test assertion and manually annotated its dependencies. The test suite contains roughly 100 test assertions in total.

Further, we exploited the fact that a test assertion and its dependencies define a new program that verifies successfully, as explained in Section 5.3 (Pruning Viper Programs). In particular, the test framework extracts all explicit assertions of a program and creates a new, pruned program for each of them. The test passes if all new programs verify successfully. The advantage of this second test strategy is that no manual annotations are required. However, the test might report false positives (see Section 5.3 (Pruning Viper Programs)). Pruning tests are executed on all unit tests, programs generated by precision benchmarks, and 9 real-world programs (available in the Silicon GitHub repository [6]) which are, for example, implementing the Gaussian sum, graph copy, or binary search on sequences.

All annotated and pruning tests pass successfully and we are currently not aware of any soundness issues in the algorithm or the implementation.

Soundness of the Gobra Dependency Analysis

Soundness of the *Gobra* dependency analysis follows from soundness of the *Viper* dependency analysis and the sound mapping from low-level nodes to *Gobra* nodes.

Unfortunately, we found two unexpected cases that lead to unsound dependency analysis results. One example included an invariant `acc(x.f)` of a for-range loop that was not represented by an assumption node at all. Instead, nodes created by this invariant got associated with the declaration of `x` as a source. This becomes apparent only when looking at the low-level dependency graph. On the other hand, the analysis result presented to the Gobra user is unsound. This issue was specific to for-range loops and was caused by an inconsistent source mapping. The second example was a global constant which got inlined into the Viper program and was not represented in the dependency graph. Both unsoundness issues were caused

by unexpected behavior in the translation from Gobra to Viper. Notably, running dependency analysis directly on the Viper program led to sound Viper dependency results.

In practice, we found that the Gobra dependency analysis is sound in most but not all cases. Future work should ensure soundness by thoroughly testing and handling all Gobra features and their encodings. In the remainder of this thesis, we limit benchmarks to Gobra programs that do not exhibit these unsound features and we manually verified soundness of the analysis for these programs.

Conclusion

Research question RQ1 seeks to answer whether the proposed dependency analysis algorithm is sound. To this end, we provided an informal soundness argument for the Viper dependency analysis. However, for Gobra programs, we found two unsound cases caused by Gobra encodings not yet covered by the dependency analysis.

8.2 RQ2: Precision

While soundness is the most important metric and guarantees correct results, good precision is also required in practice. In particular, the worst-case scenario, where dependency analysis reports everything as a dependency, does not reveal any interesting information and is thus of little use. In this section, we evaluate whether this worst-case scenario arises and how closely our analysis adheres to the definition of precision given in Section 3.4 (Soundness and Precision). To this end, we quantify precision using the following precision metric:

Definition 8.1 (Precision Metric)

$$\textit{precision } \varphi := \frac{\#\textit{real dependencies}}{\#\textit{reported dependencies}}$$

Precision φ is a number between 0 and 1 which corresponds to the fraction of reported dependencies that are indeed real dependencies. In the remainder of this thesis, we say that an analysis result is precise if $\varphi = 1$. Otherwise the result is imprecise. The term "false positives" refers to assumptions which are not real dependencies but reported as dependencies nonetheless. To compute precision φ, the reported dependencies are extracted from the dependency graph. The real dependencies, i.e., the ground truth, have to be determined by hand. Note that computing precision only makes sense if the analysis result is sound, i.e., the real dependencies are a subset of the reported dependencies.

Precision in Viper Programs

The burden of manually determining the ground truth restricts our options to conduct thorough benchmarks. To tackle this, we aim at generating a large number of Viper programs where the dependencies are known and can thus be annotated automatically. We carefully define base programs and interferences, both of which cover a large part of Viper's language features. Each combination of base program and interference defines a new, annotated program such that the ground truth can be computed automatically using the annotated test framework described in Section 7.5 (Annotated Tests). With this strategy, we generated 360 programs with roughly 1920 test assertions. The scripts to generate these programs can be found in the Silicon GitHub repository [6].

The goal of this benchmark is to exercise how various features, defined as interferences, behave in larger programs, i.e., whether they introduce imprecision. While we did our best to generate a variety of programs and used our knowledge of the algorithm to craft examples that might trick the analysis into reporting irrelevant assumptions, we do not claim that this benchmark is complete. In the following, the generation of programs is explained in more detail. Then, the results are presented and discussed.

Generating Annotated Viper Programs

The generation of test cases is based on combining base programs and interferences. We reuse the unit tests described in Section 8.1 (RQ1: Soundness) as base programs. Importantly, each unit test consists of many methods, each of which has exactly one test assertion. Additionally, we define interferences which are short code fragments exercising a specific Viper features. Later, each interference is integrated into each base program creating interesting test cases. Most statements in base programs and interferences are annotated as being a dependency or irrelevant. However, some statements do not have an annotation because they might or might not be a dependency depending on the combination of base program and interference as well as the test assertion.

The interferences are carefully integrated into and adapted to each base program such that (1) they interact with the base program as much as possible through accessing and updating its variables and heap locations, (2) the new program verifies successfully, and (3) statements marked as irrelevant are indeed irrelevant for verifying the test assertion.

Property (1) is ensured by annotating all base programs with information about which variables are available for read and write access, respectively, as well as invariants established by the base program which can thus be assumed and asserted by the interference. The interference contains place-

holders which are replaced by a suitable program variable or invariant whenever possible. Otherwise, new variables are generated and potentially, permissions are inhaled. The new program is always verified before executing the benchmark which ensures property (2). We confirm property (3) by removing all statements annotated as irrelevant and verifying the resulting program. This works similarly to pruning programs as discussed in Section 5.3 (Pruning Viper Programs).

Next, we briefly describe each interference used in this Viper precision benchmark.

1. **Inhale Invariant**: Inhales an expression that was already established by the base program, i.e., asserting this expression would succeed. Note that several of the other interferences inhale new assumptions which is why there is not a separate interference for this case.
2. **Assert Invariant**: Asserts a property already established by the base program.
3. **Assignment**: Defines a single integer assignment where the right-hand side is the addition of two integer variables, which can be pure or impure, quantified or non-quantified. We examine pure and field assignments separately:
   - 3.1. **Pure Assignment**: The receiver is a pure, integer variable.
   - 3.2. **Field Assignment**: The left-hand side is an integer field of a heap resource. Whenever possible, a (potentially quantified) heap resource of the base program is chosen. If no such reference is available, a non-quantified heap resource is generated.
4. **Disjoint Heap Resource**: Declares a new heap reference that is guaranteed not to alias any of the base program's references. Note that the statement establishing this non-aliasing guarantee is neither marked as a dependency nor as irrelevant because it can be either, depending on the test assertion. Then, it inhales access to a field for this reference and writes to it. We distinguish four cases:
   - 4.1. **Non-Quantified, Identical Field (NI)**: Inhales access to the field `f` which is the default field used by base programs.
   - 4.2. **Non-Quantified, Disjoint Field (ND)**: Inhales access to the field `gen_f` which is *never* used by any base program.
   - 4.3. **Quantified, Identical Field (QI)**: Inhales access to the field `f` for a sequence of references, i.e. it inhales quantified permissions.
   - 4.4. **Quantified, Disjoint Field (QD)**: Inhales access to the field `gen_f` for a sequence of references.

5. **Function Call**: Calls a function adding two integer inputs and returning the integer result. A precondition requires positivity of the second input and a postcondition ensures that the result is greater than the first input. The interference establishes the precondition before calling the function. We designed two variations:
   5.1. **Pure Function Call**: Both inputs are pure, integer variables. Therefore, the function has no permission requirements.
   5.2. **Impure Function Call**: One input is a heap reference for which access to the integer field `f` is required as a precondition.
6. **Method Call**: Calls a method taking one input that is required to be positive. Two postconditions guarantee that the input is incremented by one and the result is greater than one, respectively. We differentiate between pure and impure method calls:
   6.1. **Pure Method Call**: The input is an integer and the method returns the incremented integer.
   6.2. **Impure Method Call**: The input is a heap reference for which write access to the integer field `f` is required as a precondition. This field is incremented by one and full access to it is returned as a postcondition. It is important to note that whenever permission is transferred to a method and received back afterwards, every subsequent access to the corresponding heap resource depends on that method call, namely on the postcondition returning permission to the resource. This is not imprecise because methods can modify the heap, thus the postcondition returning permission is relevant. For this reason, we generate a *new* heap reference and pass it to the method call instead of reusing references defined in the base program.
7. **Predicate**: Predicates follow the same reasoning as impure method calls since they are subject to the same kind of permission transfer. Therefore, this interference generates a new heap resource whose permission is temporarily transferred to a predicate, i.e., the predicate is folded and unfolded. The predicate defines two inputs and asserts full permission to the input reference as well as positivity of the integer input.
8. **Magic Wand**: Magic wands are similar to predicates. This interference packages permission to a newly generated heap resource under the condition that a chosen integer variable is positive. The magic wand is applied later on to get back permission.
9. **Branch**: Defines one or more branches, each of which contains some assignments. We created four interferences covering different cases:
   9.1. **Feasible Branch**: All branches are guaranteed to be feasible.

9.2. **Infeasible Branch**: One branch is infeasible and contains an `assert false` statement to prove it.

9.3. **Nested Branch**: Contains nested branches. Here, some branches might be infeasible depending on the base program.

9.4. **Branch and Function Call**: Calls a function adding two positive integers inside branches. The function preconditions are ensured by branch conditions. Depending on the base program, some branches might be infeasible.

10. **Loop**: Defines a loop where every iteration decreases a loop counter and increases the result variable by amount n. The loop counter, result variable, and increment n are chosen from the base program's variables whenever possible. The first invariant defines an upper bound on the loop counter. The second invariant ensures that the result is equals to n multiplied by the number of executed loop iterations. We distinguish three cases:

    10.1. **Pure While Loop**: Operates on integer variables and thus does not require any permission constraints.

    10.2. **Impure While Loop**: The loop counter and result variable are heap resources, thus write permission to them are enforced by invariants.

    10.3. **Goto**: Defines the same behavior as the pure while loop but encoded using a `goto` statement instead of a while loop.

11. **Quasihavoc**: Consists of a single `quasihavoc` statement which invalidates the value of a heap location. We distinguish four cases:

    11.1. **Identical Reference, Identical Field (II)**: Havocs a heap resource used by the base program whenever possible.

    11.2. **Identical Reference, Disjoint Field (ID)**: Inhales access to the field `gen_f` for a heap resource used by the base program and havocs this new field.

    11.3. **Disjoint Reference, Identical Field (DI)**: Defines a new heap reference, inhales access to the field `f`, and havocs it.

    11.4. **Disjoint Reference, Disjoint Field (DD)**: Defines a new heap reference, inhales access to the field `gen_f`, and havocs it.

12. **Unrelated Code**: Defines a code fragment that does not interact with the base program, i.e., it does not access or modify any of the base program's variables or heap locations. It ensures that no infeasible branches are introduced and resources do not require aliasing checks. The latter is guaranteed by using fields which are disjoint from the ones

used by the base program. This interference contains 33 statements exercising numerous Viper features such as pure and impure assignments, branches, loops, predicates, magic wands, and so forth. It also uses quantified resources. The intention is to demonstrate whether the analysis successfully detects unrelated code fragments as such.

A major challenge of designing interferences and annotating unit tests was taking into account that some dependencies might be introduced by non-aliasing proofs. In particular, when generating new heap resources and inhaling permissions to them, some test assertions might depend on the `inhale` statement since non-aliasing needs to be guaranteed. For this reason, we enforce that all heap resources generated by interferences are guaranteed to be non-aliases of the base program's resources and the statement establishing this property is neither annotated as irrelevant nor as dependency.

Benchmark Results

Table 8.1 presents the results of the Viper precision benchmark. Each column represents a base program and each row an interference. The number in each cell reports the observed precision for the corresponding combination of base program and interference. Results with non-perfect precision are indicated by red cells. Some programs failed validation steps (2) or (3) and are excluded from the evaluation, indicated by "x". The first row (0. Baseline) corresponds to the base program with no interference. We ensured perfect precision for all of them to make comparison simpler.

Subsequently, we discuss the results and provide minimal examples for the main causes of imprecision. In code listings, we annotate the assertion of interest with "test assertion" and irrelevant assumptions that get wrongly reported as dependencies with "false positive". We use row and column identifiers when referring to a specific cell of Table 8.1, for example, the cell corresponding to "5.2 Impure Function Call" integrated into base program "B - Permissions" is denoted by 5.2B.

Pure Assumptions.

We found that pure interferences exhibit perfect precision in many cases, namely, for assignments, function calls, and method calls (Table 8.1, rows 3.1, 5.1, and 6.1). In those programs, the SMT solver detects the independence, for example, because the modified variable gets either overwritten at a later point, does not appear in any of the dependencies at all, or its value is irrelevant.

Redundant Assumptions.

Imprecision can be introduced by redundant assumptions. In these cases, the SMT solver can decide whether to use the first assumption or the redundant

	A - Inhales/Exhales	B - Permissions	C - Wildcard Permissions	D - Quantified Permissions	E - Function Calls	F - Method Calls	G - predicates	H - Magic Wands	I - Branches	J - Loops	K - Domains	L - Miscellaneous
0. Baseline	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
1. Inhale Invariant	0.96	1.00	1.00	0.92	1.00	1.00	0.82	1.00	1.00	1.00	0.96	1.00
2. Assert Invariant	0.97	1.00	1.00	0.94	1.00	1.00	0.87	1.00	1.00	1.00	0.97	1.00
3.1 Pure Assignment	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
3.2 Field Assignment	1.00	1.00	x	x	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4.1 Heap Resource (NI)	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4.2 Heap Resource (ND)	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4.3 Heap Resource (QI)	0.89	0.88	0.91	0.86	1.00	1.00	0.99	0.98	1.00	1.00	1.00	0.94
4.4 Heap Resource (QD)	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
5.1 Pure Function Call	1.00	1.00	1.00	0.95	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
5.2 Impure Function Call	1.00	1.00	x	x	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
6.1 Pure Method Call	1.00	1.00	1.00	x	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
6.2 Impure Method Call	1.00	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
7. Predicate	0.95	1.00	1.00	0.93	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8. Magic Wand	0.94	0.95	1.00	0.94	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00
9.1 Feasible Branch	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
9.2 Infeasible Branch	0.92	0.77	0.85	0.80	0.93	0.93	0.80	0.88	0.87	0.89	0.89	0.96
9.3 Nested Branch	1.00	1.00	1.00	0.88	1.00	1.00	0.92	1.00	0.93	1.00	1.00	1.00
9.4 Branch and Function Call	0.94	1.00	0.92	0.86	1.00	1.00	0.88	1.00	0.76	0.94	1.00	1.00
10.1 Pure While Loop	1.00	1.00	1.00	x	1.00	1.00	1.00	1.00	1.00	x	1.00	1.00
10.2 Impure While Loop	1.00	1.00	x	x	1.00	1.00	x	x	1.00	1.00	1.00	1.00
10.3 Goto	1.00	1.00	1.00	x	1.00	1.00	1.00	1.00	1.00	x	1.00	1.00
11.1 Quasihavoc (II)	0.90	0.74	x	x	1.00	1.00	0.99	0.96	1.00	1.00	0.96	0.96
11.2 Quasihavoc (ID)	1.00	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
11.3 Quasihavoc (DI)	0.87	0.68	0.78	0.69	1.00	1.00	0.98	0.96	1.00	1.00	1.00	0.94
11.4 Quasihavoc (DD)	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
12. Unrelated Code	1.00	1.00	1.00	0.95	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 8.1: The results of the Viper precision evaluation. Each cell corresponds to the computed precision, with 1.00 indicating perfect precision. Programs that failed validation steps (2) or (3) are indicated by "x".

one. Consider Listing 8.1 where the inhale on line 6 is sufficient to prove the assertion on line 7 on any path. However, when b evaluates to true, both `inhale` statements (line 4 and 6) are executed. It may happen, and this was observed in practice, that the inhale on line 4 is reported as a dependency on this path. Since both paths (b does or does not hold, respectively) are taken into account, dependency analysis returns both `inhale` statements as dependencies of the assertion on line 7, leading to an imprecise result. Interference "1. Inhale Invariant" was subject to this cause of imprecision,

for example, seen in cells 1A and 1G in Table 8.1, page 85.

```
1 method redundantAssumption(a: Int, b: Bool)
2 {
3     if(b){
4         inhale a >= 0 // false positive
5     }
6     inhale a >= 0
7     assert a >= 0     // test assertion
8 }
```

Listing 8.1: A Viper program demonstrating how redundant assumptions can lead to imprecise analysis results.

Assert Invariant with Conjuncts.
An interesting case found for the "2. Assert Invariant" interference (e.g., cells 2A and 2K in Table 8.1, page 85) revolves around assertions with conjuncts. In Listing 8.2, the assertion on line 7 is expected to have exactly one dependency, namely to the inhale on line 2. However, dependency analysis also reports a dependency to line 3. To understand why, we need to look at the assertion on line 5 which adds the internal assumption $a \geq 0$ to the path condition, as briefly addressed in Section 6.3 (Hiding Silicon Implementation Details). In the low-level dependency graph, line 7 depends on line 5 but this dependency is not exposed to the user because internal assumptions are hidden. Instead, line 2, which was used to prove the assertion on line 5, is reported. Unfortunately, line 3 is also a dependency of line 5 and thus also reported.

The underlying issue is that `assert` statements are currently not split into their top-level conjuncts. This is left for future work. For the time being, users can circumvent this problem by splitting such assertions manually.

```
1 var a: Int, b: Int
2 inhale a > 0
3 inhale b > 0    // false positive
4
5 assert a >= 0 && b > 0 // internal dependency
6
7 assert a >= 0   // test assertion
```

Listing 8.2: A Viper program containing an assertion with conjuncts that leads to imprecision.

Field Assignments.
Interference "3.2 Field Assignment" exhibits high precision indicating that

our improvements to hide implementation details of field assignments, namely the fact that a new chunk for the left-hand side is inhaled, has its desired effect. In regards to permission to the assigned resource, neither the assignment nor dependencies of the right-hand side are mistakenly reported as dependencies.

On the other hand, imprecision was found for some assignments to *quantified* heap resources, namely for base program "D - Quantified Permissions", as shown in Table 8.1, page 85. In Listing 8.3, the `exhale` statement is expected to depend only on the two `inhale` statements. However, the analysis reports a false-positive dependency to the assignment of `idx` on line 2. The underlying reason is that the exhale internally depends on the assignment to `xs[idx].f` on line 5, more concretely, on inhaling full permission to that resource. While this internal assumption is not reported to the user, its dependencies are, including line 2 required to verify that `idx` stays within the sequence's range. To resolve this imprecision, dependencies related to the left-hand side of field assignments have to be added more carefully. This is left for future work.

```
1 method quantifiedFieldAssign(a: Int, xs: Seq[Ref]){
2     var idx: Int := 0   // false positive
3     inhale |xs| > 2
4     inhale forall x: Ref :: x in xs ==> acc(x.f)
5     xs[idx].f := a      // internal dependency
6
7     exhale acc(xs[0].f) // test assertion
8 }
```

Listing 8.3: A quantified field assignment introducing imprecision.

Heap Resources.
Apart from assignments, precision perceived in interferences using references and fields differs depending on whether the fields are overlapping or disjoint from the base program's fields. Disjoint fields make non-aliasing proofs trivial since they never point to the same heap location, even if their receivers are identical, i.e., for a reference `x` with fields `f` and `g`, `x.f` and `x.g` refer to disjoint heap locations. Therefore, interferences using different fields than the base program exhibit better precision than interferences using the same fields. This can be seen in Table 8.1, page 85, when comparing the results for interferences 4.3 and 4.4, 11.1 and 11.2, 11.3 and 11.4, respectively.

The programs indirectly tested the precision of heap summary by inhaling permission to newly generated (potentially quantified) heap resources. We did not find an example where imprecision could be attributed to heap summary, unless wildcards or permission variables are used, as briefly mentioned in Section 6.2 (Increasing Precision in the Low-Level Dependency

Graph).

Permission Transfers.
Impure method calls, predicates, and magic wands are subject to the same kind of permission transfer. In the corresponding interferences, imprecision is observed for some test assertions requiring non-aliasing proofs, e.g., for an assertion `perm(x.f)> 1/2`. In such cases, the method call, `unfold`, or `apply` statement may get reported as a false positive because the permission gained from this operation is used to prove non-aliasing. Further noise may be added because the test assertion inherits the dependencies of this operation, for example, to the associated `fold` statement. This can be a problem because it leads to a cascade of false positives. This type of imprecision was encountered, for example, in cells 7A and 8B in Table 8.1, page 85.

However, Silicon often establishes non-aliasing early on and explicitly adds the corresponding assumption, e.g., `x != y` for two references `x`, `y` where permission to field `f` would add up to more than one. The analysis then reports this assumption as a dependency instead of relying on permission gained from later operations, such as predicate unfolds. We observed this, for example, in cell 6.2A for which analysis was precise even though it contains a pattern similar to the one causing imprecision in 7A and 8A for the same test assertion.

Interestingly, the magic wand interference experienced more imprecision than the predicate and method call interferences. Investigating the cause of this observation is left for future work.

Branches.
Branches pose an interesting case because precision highly depends on whether infeasibility is introduced. Table 8.1, page 85, shows that interference "9.1 Feasible Branch" is *precise* in all but one case (where imprecision is caused by quantified permissions), while "9.2 Infeasible Branch" is always *imprecise* to some extent. This imprecision is a direct consequence of adding dependencies to the proof of infeasibility to *all* assertions on the infeasible path.

An example is depicted in Listing 8.4 where the assertion on line 9 should clearly only depend on the assignment $a := 0$ on line 8. However, since the condition on line 5 introduces an infeasible branch, the proof of infeasibility, consisting of the precondition and branch condition, is wrongly reported as a dependency.

Quasihavoc.
`Quasihavoc` statements introduce imprecision in two ways, both have been found for interferences 11.1 and 11.3 which contain `quasihavoc` statements

```
1 method infeasibility(n: Int)
2   requires n >= 0 // false positive
3 {
4   var a: Int
5   if(n < 0){      // false positive
6     a := 1
7   }
8   a := 0
9   assert a >= 0   // test assertion
10 }
```

Listing 8.4: A Viper program where exploring infeasible paths leads to imprecision.

with fields identical to base program fields. First, the `quasihavoc` statement is reported as a dependency for all subsequent accesses to the havoced resource. This is because, similarly to field assignments, `quasihavoc` is implemented as exhaling and inhaling full permission to the havoced resource. Second, due to aliasing checks executed to check which resources to havoc, the `quasihavoc` statement itself might depend on statements inhaling permission to irrelevant resources. The second case only applies to heap resources using the same field as the havoced resource. This explains the different results encountered in Table 8.1 for interferences 11.1 and 11.2 as well as 11.3 and 11.4.

Assumptions on Upper Bounds of Permissions.

The results confirm that unrelated code, i.e., row 12 in Table 8.1, is detected as such since perfect precision was achieved in all but one cases. The exception, namely cell 12D in Table 8.1, is an edge case related to the assumptions on upper bounds of permissions, illustrated in Listing 8.5. Here, the test assertion verifies non-aliasing of two references where full permission to `xs[0]` and read permission to `y` are held. Unexpectedly, the unrelated `apply` statement in line 8 assumes an upper bound on the permission of `y`, e.g., `perm(y.f)<= W`. In theory, upper bounds on permissions can be assumed at any point since holding more than full permission is not permitted anyways. This implementation detail is unfavorable because the assumption, whose source is the `apply` statement in our case, is reported as a direct dependency of the test assertion. As a consequence, dependency analysis reports the `apply` statement and all of its dependencies as false positives. This type of imprecision could be resolved in future work by marking such assumptions as internal and closed.

Results form Manually Crafted Examples

In addition to the generated test cases, we used our understanding of the dependency analysis algorithm and its implementation to manually craft

```
1 method implicitUpperBounds(xs: Seq[Ref], y: Ref)
2   requires |xs| > 2
3   requires forall x: Ref :: x in xs ==> acc(x.f)
4   requires acc(y.f, 1/2)
5 {
6   var z: Ref
7   inhale true --* acc(z.f)  // false positive
8   apply  true --* acc(z.f)  // false positive
9
10  assert xs[0] != y    // test assertion
11 }
```

Listing 8.5: A test assertion that depends on an irrelevant apply statement due to implicit bounds assumed during the apply.

some interesting examples (available in the Silicon GitHub repository [6]). Here, we discuss results found through such examples.

Dependencies across Program Components.
The setup of the annotated tests did not allow to compute precision for dependencies across program components. Manually crafted programs revealed that dependency analysis reliably detects *which* postconditions of methods and functions an assertion depends on, i.e., it does not simply report all them.

State Consolidation.
State consolidation is still imprecise in some cases. This is due to an implementation detail where assumption labels conditionalize larger parts of the composed assumption than they should. Thus, some assumption labels are reported as false positives. We expect this issue to be resolved once the implementation is fixed.

Non-Unique UNSAT Core in Loops.
We already briefly mention that redundant assumptions can cause imprecision. This can also occur in loops, as depicted in Listing 8.6. The second invariant `res >= 0` could be proven using the loop condition and the assignment to `res` on line 9. However, it is also possible to use the first invariant instead of the loop condition. While both options are valid, the second one leads to imprecision since all dependencies of the invariant are inherited, for example, the assignment to `i` in the loop body.

```
1 method nonUniqueUnsatCore(){
2   var i: Int, res: Int
3   i := 10             // false positive
4   res := 0
5   while(i > 0)
6     invariant i >= 0  // false positive
7     invariant res >= 0
8   {
9     res := res + i
10    i := i - 1        // false positive
11  }
12  assert res >= 0     // test assertion
13 }
```

Listing 8.6: A Viper program demonstrating how non-uniqueness of UNSAT cores can lead to imprecise analysis results.

Superfluous Permission Inhales.

Dependency analysis tracks the entire modification history of each heap resource. However, some modifications might be unnecessary and are thus false positives; for example, the second inhale in Listing 8.7 is not required for proving the test assertion but is reported as a dependency nonetheless.

```
1 inhale acc(x.f, 1/2)
2 inhale acc(x.f, 1/2)  // false positive
3 exhale acc(x.f, 1/2)  // test assertion
```

Listing 8.7: An example demonstrating imprecision caused by unnecessarily inhaling more permission to a heap resource.

Incompleteness of SMT Solvers.

SMT solvers are incomplete and can thus not prove all valid assertions. This can lead to imprecision in the dependency analysis, as demonstrated by Listing 8.8. The quantified path condition is equivalent to false but the SMT solver cannot prove this. Consequently, both paths are explored and dependencies to the assumptions in both branches are reported, even though one of them is effectively unreachable. We cannot eliminate this type of imprecision in the dependency analysis as it is caused by SMT solver incompleteness.

Precision in Gobra Programs

To evaluate precision of the Gobra dependency analysis, we manually analyzed several Gobra programs. In theory, the Viper precision benchmarks

```
1 var a: Int
2 if (exists x: Int, y: Int, z: Int :: x>0 && y>0 && z>0 && x
      *x*x + y*y*y == z*z*z) {
3     // effectively unreachable code
4     assume a == 0 // false positive
5 } else {
6     assume a == 0
7 }
8 assert a == 0 // test assertion
```

Listing 8.8: A Viper program with a path condition whose unsatisfiability cannot be proven by the SMT solver.

could be lifted to Gobra for a more thorough evaluation. However, Gobra currently does not support annotations on statements and expressions, which is why we leave this extension for future work.

For our evaluation, we use programs from the Gobra evaluation suite described in Wolf et al. [33]. They cover a variety of Gobra features, some of which are currently not supported by the dependency analysis. Therefore, some programs are skipped, most importantly, the ones using interfaces and concurrency. We provide the chosen benchmark programs in the Gobra GitHub repository [4]. Line numbers mentioned in the following discussion always refer to these programs.

For each Gobra program, we manually evaluated precision of selected assertions. Additionally, we explored the dependencies of some assertions coming from VerifiedSCION's `addr` package [3]. We chose a variety of assertions, in particular, some with few expected dependencies and some covering a large fraction of the program.

The results are listed in Table 8.2, page 93, which presents the number of reported and real dependencies as well as the resulting precision (computed according to Definition 8.1, page 79). Further, we manually determined the number of lines of code (LOC) of each program to give an idea on how many assumptions were not reported as dependencies. However, note that a single source code line may contain several assumptions, namely when having top-level conjuncts, or no assumption, for example, when it is an explicit `assert` statement. Lastly, the last column indicates the main cause of imprecision, which are discussed next.

Let us first mention some high-level observations. We found that Gobra's variable declarations differ from Viper's since they implicitly initialize the variable. This creates an assumption node for each variable declaration. Further, unexpected sources were encountered; for example, a function parameter `a` of type `int` might introduce some assumptions with source

Line	#Reported	#Real	Precision	Imprecision Cause
Relaxed Prefix (48 LOC)				
12	6	2	0.33	infeasibility, declarations
14	34	14	0.41	loop invariants
16	25	17	0.68	declarations
29	6	2	0.33	infeasibility
31	11	8	0.72	infeasibility
36	23	3	0.13	reordering invariants
37	13	9	0.69	infeasibility
Zune (34 LOC)				
36	8	6	0.75	declarations
37	9	6	0.67	declarations
38	2	1	0.50	declarations
58	3	3	1.00	-
Dutchflag (34 LOC)				
7	21	2	0.10	slice update
8	17	14	0.82	declarations
9	33	21	0.64	slice update
10	28	24	0.86	declarations
11	35	24	0.69	invariants, slice updates
Example 2.1 (16 LOC)				
7	8	2	0.25	for-loop, declarations
8	14	11	0.79	declarations
24	22	1	0.05	slice and internal encodings
26	30	15	0.50	internal encodings, declarations
Binary Search Tree (211 LOC)				
167	10	3	0.30	internal encodings, declarations
302	16	11	0.69	declarations
304	70	50	0.71	internal encodings, declarations
306	160	117	0.73	internal encodings, declarations

Table 8.2: The results from manual precision analysis on programs from the Gobra suite.

`a` and some with source `a int`, i.e., it is represented twice. Additionally, some internal encodings, for example artifacts from array initializations, are mistakenly exposed to users even though they should be marked as internal. In Table 8.2, we denote these cases by "declarations" and "internal encodings", respectively. Further, we found patterns that were already revealed in the Viper precision benchmarks, in particular, caused by infeasible branches. Next, some interesting examples discovered in each Gobra program are discussed.

Relaxed Prefix checks whether one of the input slices is a prefix of the other

but accepts one non-matching element. None of the two slices is modified and we did not encounter any imprecision caused by these quantified heap resources. However, most assertions report two irrelevant dependencies due to an infeasible path. An interesting case is the invariant on line 37 asserting boundary constraints on the output variable for which dependency analysis is extremely imprecise ($\varphi = 0.14$). The underlying reason was that this invariant unexpectedly depended on the proof that another invariant *inv* is preserved. Invariant *inv* has itself numerous dependencies, all of which are inherited by line 37 resulting in poor precision. Interestingly, we discovered that reordering these two invariants resulted in significantly more precise results. This can be explained by the fact that, due to reordering, the invariant on line 37 is now proven before *inv* and can hence not depend on *inv*'s proof.

Zune is a small, pure program, i.e., it does not access the heap. Here, only few false positives are reported, most of which are declarations. We observed that irrelevant assignments and loop invariants were detected as such. Notably, dependency analysis detects that none of the queried assertions depends on the only branch condition or on the body of a helper function, which indicates good precision.

Dutchflag operates on slices which introduce imprecision caused by element updates; for example, the postcondition granting permission to all slice elements (line 7) depends on a slice update. We hypothesize that the underlying reason for this behavior is similar to field assignments in Section 6.2 (Increasing Precision in the Low-Level Dependency Graph) and might be resolvable using a similar technique. By depending on slice updates, dependencies to the index boundary conditions defined as invariants are inherited. This cascading effect explains the low precision for line 7.

In Example 2.1, we noticed that some reported dependencies are artifacts of internal encodings that should not be exposed to the user, for example, calls to the box and unbox functions of the slice encoding. This is also the reason why for lines 24 and 26 the number of reported dependencies is significantly higher than the manually determined LOC of the program. Line 24, which asserts that the part of the slice not being passed to the preceding function call remains unchanged, exhibits poor precision ($\varphi = 0.05$). From Gobra's perspective, this part of the slice should not depend on the function call but due to slice encodings, it is reported as a dependency nonetheless. Additionally, dependencies coming from that function's body are inherited causing a cascade of false positives. This explains the low precision and indicates that encodings of Gobra features need to be explored more thoroughly in future work.

Binary Search Tree defines a tree structure and an associated predicate which encapsulates permission to all nodes and states that the nodes are sorted. All methods operating on the tree require this tree predicate as a precondition,

operate on the unfolded predicate, and return the folded predicate to the caller. A function `client0`, shown in Listing 8.9, creates a new tree, inserts an element, and deletes it. After each of these operations, an assertion ensuring the expected behavior is executed. They correspond to assertions on lines 302, 304, and 306 in the original source code. Here, the call to `Delete` depends on the call to `Insert` since the tree predicate provided by the postcondition is required. However, it only depends on some but not all of `Insert`'s postcondition, indicating good precision. In particular, the assertion on line 304 depends on more postconditions of the `Insert` function than line 306 does. This is expected because correctness of the `Insert` method is irrelevant for line 306, but not for line 304. Unfortunately, line 306 still depends on most of `Insert`'s body since most statements are relevant for folding the tree predicate before returning it, which is a pattern we saw in almost all functions of Binary Search Tree.

```
1 func client0(value int) (t *Tree) {
2     t = NewTree()
3     assert !t.pureContains(value, 2)  // line 302
4     t.Insert(value)
5     assert t.sortedValues() == seq[int]{value} // line 304
6     t.Delete(value)
7     assert !t.pureContains(value, 2)  // line 306
8     return t
9 }
```

Listing 8.9: The client function of Binary Search Tree used for the precision benchmarks.

The results encountered in VerifiedSCION's `addr` package (available on GitHub [5]) are similar. In particular, we found several examples where an assertion depends on some but not all postconditions of a called function, which is a good indicator that it does not correspond to the worst-case scenario. Further, the analysis successfully reports dependencies across many source files. For example, the dependency set of `host.go` line 478 contains function postconditions from `slices.gobra` and `ip.gobra`. The dependencies are uniquely identifiable by their source file and line number, which is stored in all nodes as part of the source. However, during verification of the `addr` package only the function specifications of these Gobra files are imported and, thus, dependencies to function bodies are not found, i.e., they are treated as trusted functions without bodies.

Conclusion

The answer to the second research question (RQ2) is that dependency analysis is not fully precise. Experiments determining quantitative precision of

various test cases reveal examples with good, or even perfect, precision. Dependency analysis can detect irrelevant code as such, for example, irrelevant (field) assignments, branch conditions, or loop invariants. In other words, the worst-case scenario where everything is reported as a dependency was not encountered. In particular, dependency analysis reports which postconditions of which functions are relevant, i.e., dependencies to some but not all postconditions are reported. Further, statements inhaling permission to references which are not exercised by the tests assertions are correctly identified as irrelevant in many cases.

However, we also found examples exhibiting poor precision. Common patterns causing imprecision include, among others, superfluous infeasibility proofs, redundant assumptions, non-unique UNSAT cores, and assertions that cannot be proven due to SMT solver incompleteness. Imprecision is further caused in impure programs by the need of non-aliasing proofs and implementation details of the symbolic heap, such as quantified field assignments and state consolidation.

Unsurprisingly, imprecision found in Gobra programs has similar causes as the ones discovered in Viper benchmarks, in particular, caused by infeasible branches and redundant assumptions. In addition to that, we discovered that Gobra variable declarations cause unexpected noise and that slices are probably subject to the same implementation detail causing imprecision as field updates. Since we covered only a small fraction of the Gobra features, we expect to find more examples like that in the future. Overall, lifting the analysis to Gobra worked rather well for a start but future work is required to explore more Gobra features.

Moreover, we observed cascades of false positives, i.e., one false positive introducing more noise because additional dependencies are inherited through the false positive. To help users in the analysis process, it might be beneficial if user tools provide the option to label a reported dependency as a false positive and re-computing the dependencies with this additional information. This helps filtering out irrelevant dependencies when querying dependency graphs.

Overall, we argue that the dependency analysis is already precise enough to reveal insightful information but precision should be improved in future work.

8.3 RQ3: Performance

A sound and precise dependency analysis is of little use when construction of the dependency graph incurs significant runtime. Thus, another important metric of the analysis is its performance. There are two interesting aspects. First, the impact on verification time indicates whether the dependency graph

can be constructed within reasonable time. Second, the user experience for analyzing dependency graphs heavily depends on the response time of queries in the command-line tool. In the following, we present benchmarks executed measuring these metrics. All performance benchmarks were executed on a Windows 11 system with a 1.8 GHz 4-Core Intel Core i7 CPU, 16 GB of RAM and OpenJDK 11.

Impact on Verification Time

To assess the impact on verification time, we benchmark the Gobra programs and VerifiedSCION `addr` package which were already discussed Section 8.2 (RQ2: Precision). While we also benchmarked the verification time of Viper programs, the differences were negligible due to the small size of Viper programs, thus we omit the results. Gobra programs were verified 12 times and we report the average after removing the longest and shortest verification time. Due to the long runtimes, the VerifiedSCION `addr` package was only verified three times and all measurements are kept for computing the average.

We suspect that the execution of infeasible paths has significant impact on verification time, in particular, because it can cause path explosion. Therefore, we compare three different Silicon configurations:

1. The baseline corresponds to verification without dependency analysis.
2. Analysis refers to verification where dependency analysis is enabled and all (in)feasible paths are executed, i.e., this is the sound dependency analysis.
3. "Analysis with infeas checks" is an unsound version of the dependency analysis where infeasible paths are not executed.

The difference between (2) and (3) illustrates the impact of executing infeasible paths.

The results of the described Gobra benchmarks are visualized in Figure 8.1 which shows the mean verification time measured for various programs with the aforementioned Silicon configurations and the standard deviation as an error bar. Additionally, we provide the dependency graph sizes in Table 8.3 and the raw results in the Appendix, Table 1, page 122.

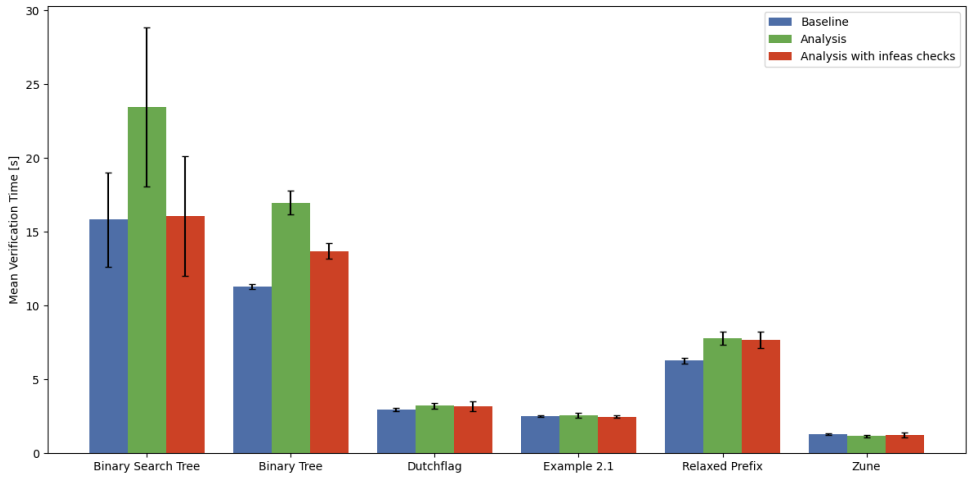

Figure 8.1: The verification times of Gobra evaluation programs with varying Silicon configurations. Note that analysis with infeasibility checks (red) is unsound.

Program	#Assumptions	#Nodes
Binary Search Tree	311	320
Binary Tree	85	88
Dutchflag	52	66
Example 2.1	27	53
Relaxed Prefix	60	76
Zune	40	46
VerifiedSCION addr package	1203	1380

Table 8.3: The dependency graph sizes of Gobra programs used for the performance benchmarks.

Overall, the baseline has the shortest verification time and sound dependency analysis the longest, which was expected. However, the impact of infeasible paths differs depending on the program and they are not the only factor that slows down verification, as seen in Binary Tree. For small Gobra programs, the impact on verification time was negligible. Medium-sized programs (Binary Search Tree and Binary Tree) incur larger overheads but verification time is still reasonable.

Table 8.4 presents the verification runtime for the VerifiedSCION `addr` package. In this large-scale program, dependency analysis performs poorly: Verification time increased by a factor of 28. Executing infeasible paths does not explain this poor performance, as indicated by the fact that the unsound version of dependency analysis is also significantly slower than the baseline.

There are various factors that could explain this observation. First, SMT

Silicon configuration	Avg runtime	Std dev
Baseline	57s	1.7
Dependency analysis (sound)	1585s	240
Analysis with infeas checks (unsound)	1231s	13

Table 8.4: The verification times measured for the VerifiedSCION `addr` package using Silicon with different configurations.

solver queries could be slowed down since dependency analysis prevents Silicon from simplifying internal terms and complicates some assumptions by adding conditions on assumption labels. Second, graph operations like lifting graphs to Gobra or joining graphs collected for different program components have an impact on verification runtime that scales with the size of the low-level dependency graph. We leave profiling and performance optimizations for future work.

Performance of Dependency Graph Queries

Next, we measure the response time of graph queries invoked through the command-line tool discussed in Section 7.3 (Querying the Graph). We use the same Gobra programs as before and the VerifiedSCION `addr` package. Once again, results measured on Viper programs are omitted because the response times for these small programs are short and do not provide additional insights. We only benchmark the dependency set query (query 1) since it is the most important one. Other queries are expected to reveal similar results since they are implemented analogously. Each query is executed 12 times but the slowest and fastest runs are removed before computing the average.

The results are provided in Figure 8.2 where each data point represents one query. We show the mean response time and the standard deviation in correlation to the number of reported dependencies. Raw results can be found in the Appendix, Table 2, page 123.

Encouragingly, all measured response times were below two seconds, even for queries in large graphs. However, the results for Binary Search Tree suggest that the response time scales with the number of reported dependencies.

Note that the query is executed on the low-level dependency graph which contains more details than what is reported to the user, i.e., it contains all low-level nodes before they are merged into Gobra nodes. The two queries with the longest response time, both from Binary Search Tree, computed 370 and 673 low-level dependencies before merging the identified nodes into 70 and 130 Gobra nodes, respectively. When plotting the response times against the size of such low-level dependency sets, the graphs look similar and are thus omitted. However, the raw data is included in Table 2, page 123, in the appendix.

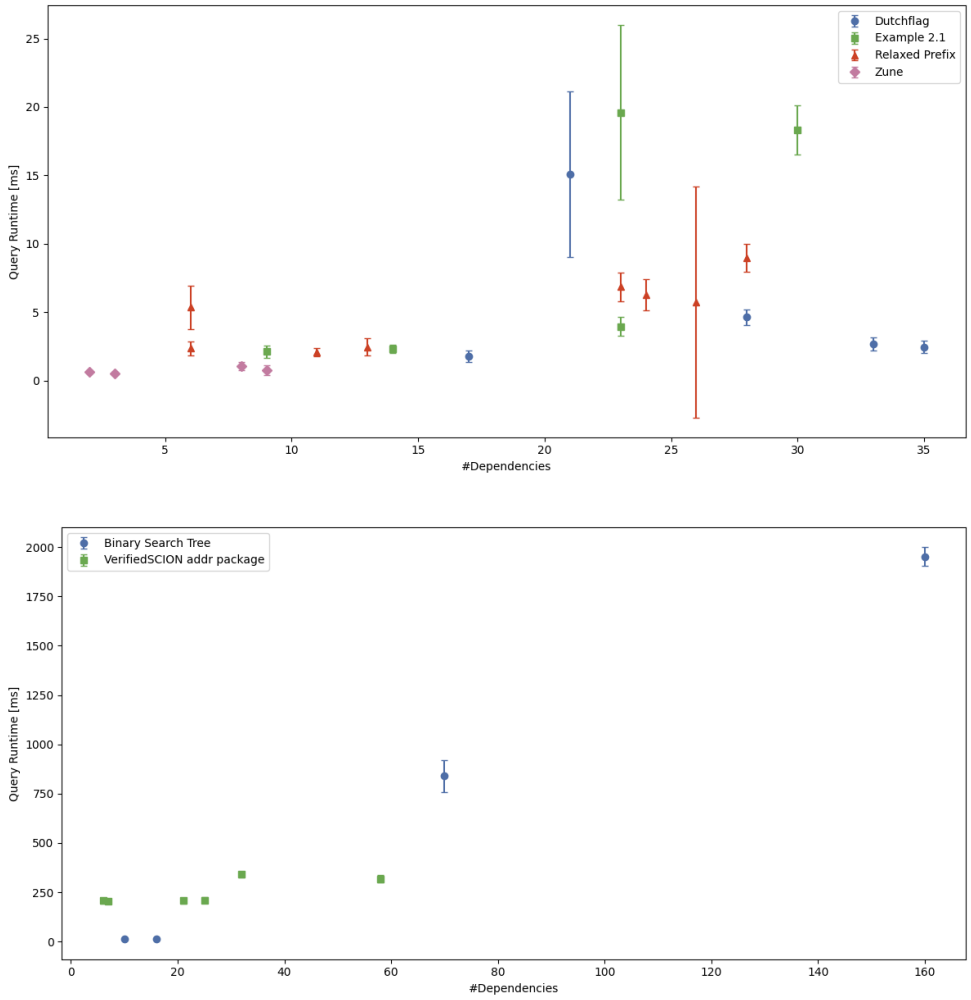

Figure 8.2: The response times of dependency set queries of various assertions from Gobra programs and VerifiedSCION's addr package.

Conclusion

To answer the third research question (RQ3), we executed performance benchmarks on Gobra programs. For smaller programs, the verification time is still reasonable but in real-world projects like VerifiedSCION, dependency analysis slows down verification significantly. Hence, the dependency analysis scales poorly to large-scale programs at the moment. On a positive note, graph queries invoked on the final dependency graph are fast enough to provide a satisfiable user experience for the command-line tool once the graph is available, even for larger graphs.

8.4 RQ4: Proof Coverage

Proof coverage, defined in Section 5.2 (Proof Coverage), was a direct application of the dependency analysis. It is thus not surprising that the accuracy of proof coverage is a direct implication of soundness and precision of the analysis.

Let us for now assume that the set of all assumptions A, namely the denominator of the proof coverage formula, contains all assumptions but no additional noise. In other words, it is complete and minimal. Then, a sound analysis, guaranteeing that all existing dependencies are reported, provides an upper bound on proof coverage, i.e., the real proof coverage is smaller or equal to the reported coverage. On the other hand, a precise analysis, which never reports irrelevant dependencies, corresponds to a lower bound on proof coverage, i.e., the real proof coverage is at least what is reported. For our dependency analysis, this implies that low proof coverage indicates incomplete specification in the sense that not all assumptions are covered, while high proof coverage does not imply good specification because it could be caused by imprecision.

Unfortunately, in practice, minimality of the set of all assumptions A is not (yet) guaranteed. The main noise in Gobra programs was caused by declarations. Declarations should always be treated as being covered but, due to their encoding, (potentially uncovered) assumptions are introduced nonetheless. As a result, the reported proof coverage in Gobra program does neither provide an upper nor a lower bound on the real proof coverage. In future work, noise caused by declarations should be filtered out such that an upper bound can be guaranteed. To circumvent this in the meantime, our command-line tools presents all uncovered nodes such that users can estimate the accuracy and adjust proof coverage accordingly.

Table 8.5 provides proof coverage of various Gobra *functions*, computed according to Definition 5.7, page 48. We also provide the number of uncovered nodes revealed by the same query and the real coverage which was estimated manually. To see the impact of noise caused by declarations, we also provide the "cleansed" proof coverage which is computed as

$$\text{cleansed cov.} = \frac{\#\text{covered nodes}}{\#\text{assumptions} - \text{noise}}$$

where noise is the number of uncovered assumptions originating from Gobra declarations.

The results reveal that the functions from the Gobra evaluation suite achieve high proof coverage, which is expected because these programs have been thoroughly specified. However, the reported coverage is too low in most cases, which can be accounted to the aforementioned noise introduced by

Gobra Function	Reported Cov.	#Uncovered Nodes	Cleansed Cov.	Real Cov.
Relaxed Prefix	0.88	7	1.0	0.89
Zune				
convertDays	0.35	11	0.6	0.5
convertDaysWithInvariants	0.68	7	0.93	0.90
Dutchflag	0.79	11	1.0	0.96
Example 2.1				
client	1.0	0	1.0	1.0
incr	0.83	3	1.0	1.0
Binary Search Tree				
NewTree	0.78	2	1.0	1.0
Insert	0.88	2	1.0	1.0
Contains	0.75	5	1.0	1.0

Table 8.5: The reported and real proof coverage of Gobra evaluation programs.

declaration encodings. This claim is supported by the cleansed proof coverage results. For Relaxed Prefix, Dutchflag, and Zune, cleansed coverage is higher than real coverage due to imprecision in the computation of the dependency sets, i.e., uncovered nodes are wrongly reported as covered.

Interesting is that the two functions in Zune are identical except that they differ in the completeness of their specification. This difference is reflected in the reported, cleansed, and real proof coverage, which indicates that proof coverage is suitable to estimate the completeness of the specification, i.e., high coverage indicates thorough specification.

So far, we computed proof coverage of Gobra *functions*. Similarly, proof coverage results for Gobra *assertions* (Definition 5.6, page 47) are reported in Table 8.6. The real coverage is computed using the number of real dependencies as reported in Table 8.2, page 93, and the number of real assumptions, which was determined manually. We omit assertions having dependencies across multiple functions since coverage is limited to one function.

Results show that the reported coverage can be higher or lower than the real coverage which is caused by noise and imprecision as explained previously. On the other hand, the cleansed coverage is higher than real coverage in all cases. Two highly inaccurate coverage results are found for Dutchflag, line 7, and Example 2.1, line 7. We already observed that these assertions are subject to high imprecision in Table 8.2, page 93. On the other hand, assertions with high precision, such as lines 8 and 10 of Dutchflag, report more accurate proof coverage.

Line	#Reported Cov.	#Uncovered Nodes	Cleansed Cov.	Real Cov.
Relaxed Prefix				
12	0.12	50	0.19	0.04
14	0.61	22	0.71	0.30
16	0.47	30	0.57	0.37
29	0.12	50	0.19	0.04
Zune				
36	0.45	12	0.63	0.46
38	0.14	19	0.21	0.08
Dutchflag				
7	0.42	30	0.53	0.06
8	0.38	32	0.49	0.41
9	0.63	19	0.80	0.62
10	0.56	23	0.71	0.71
11	0.67	17	0.85	0.71
Example 2.1				
7	0.56	9	0.64	0.17
8	0.83	3	1.0	0.92

Table 8.6: Proof coverage computed for selected assertions of Gobra programs.

Conclusion

Referring back to RQ4, proof coverage is not yet accurately computed. The main issues are imprecision in the computation of dependency sets, which results in reporting too high coverage, and noise present in the set of all assumptions, which results in reporting too low coverage. We thoroughly discussed the former in Section 8.2 (RQ2: Precision). The latter is caused by encoding details of Gobra declarations. Hiding these implementation details would result in more accurate proof coverage results and more importantly, would guarantee that the reported proof coverage provides an upper bound. This is left for future work.

8.5 RQ5: Case Study on Analyzing the Impact of Bugs

We motivated this thesis partly by claiming that dependency analysis is able to estimate the impact of (potentially invalid) assumptions. To support this claim, we present a case study executed in the context of Viper and Gobra programs.

Let us, once again, look at the Binary Search Tree program but this time the code contains a bug in the `delete` method (see Listing 1, page 120, in the appendix). More concretely, the value is not correctly deleted on some paths and the postcondition stating that the resulting tree does not contain

the value anymore fails to verify. The goal is to identify all assertions in any program component that depend on this invalid postcondition and are thus affected by the bug.

Dependency analysis successfully reveals that this postcondition is required to ensure the postcondition of the `Delete` method and by a client asserting that the deletion was successful. Moreover, it correctly detects that it is *not* a dependency of the same client's postcondition, which ensures access to the tree. While this postcondition depends on the call to the `delete` method and some of its postconditions, namely the one giving back access to the tree, it does not depend on the erroneous postcondition. We conclude that the analysis can distinguish between assertions affected by a bug and assertions that are not. However, note that false positives might be reported due to imprecision in the dependency analysis. The same process can be applied to postconditions of unverified methods and functions and, more generally, to all assumptions.

One might argue that extracting this information is possible without dependency analysis, namely by removing assumptions, re-running verification, and analyzing the verification results. However, with dependency analysis, re-verification becomes obsolete and analyzing the impact of multiple assumptions requires only one verification run.

Note that we successfully used the dependency analysis to extract information about a *partially* verified program. We go a step further and extract information about a *method* containing verification failures. For this purpose, verification is executed with the Silicon configuration option `numberOfErrorsToReport=0`. With this configuration, verification does not abort when an assertion fails. Instead, it reports the error, explicitly assumes the failed assertion, and continues verification. Thus, failing assertions can act as (explicit) assumptions under this configuration.

By extracting all dependents of assumptions added in this manner, dependency analysis reveals subsequent assertions that might fail when removing the assertion or, phrased differently, that would benefit from establishing this assertion. A minimal example is provided in Listing 8.10 where verification of the assertion on line 4 fails. The analysis reports a dependency between the assertion and the assignment on line 6.

This might be an interesting use case of dependency analysis in assertion-based debugging [13] where users add assertions to break verification failures into smaller pieces. However, assuming a fact that failed to be proven can easily introduce a contradiction making the path infeasible. Consequently, dependency analysis potentially adds many dependencies to the infeasibility proof creating a lot of noise. In future work, one could extend this by exploring how insightful this process is in real-world programs or potentially

```
1 var a: Int, b: Int
2 inhale a >= 0
3
4 assert a > 0 // fails
5
6 b := b/a // requires a != 0
```

Listing 8.10: A Viper program with a failing assertion. The analysis reports a dependency between the assignment and the assertion if run with `numberOfErrorsToReport=0`.

adding features to the dependency analysis that support this use case more thoroughly.

Conclusion

For the last research question (RQ5), we conclude that dependency analysis is applicable to partially verified programs where it reveals, for example, which assertion depend on currently unverified postconditions. Further, when run with the Silicon configuration option `numberOfErrorsToReport=0`, failed assertions are turned into explicit assumptions whose dependents can be extracted easily, revealing whether establishing the assertion helps resolving subsequent verification failures.

Chapter 9

Related Work

In this chapter, we discuss prior work related to our main contributions or use cases of our dependency analysis.

Dependency Definitions and Analyses

Analyzing dependencies and building dependency graphs of programs are not new concepts. In compiler design, they appear, for example, in control-flow, data-flow, reaching definitions, and pointer-alias analysis [7,17,19,24]. However, the problem domain and the definition of what a dependency is in compiler design analysis differs to ours which is related to program verification.

A definition of dependency more similar to ours is provided in the context of model checking where an inductive validity core [20] is defined as the part of the model required to establish a safety property. Similarly to our approach, it exploits UNSAT cores of SMT solvers and the extracted dependencies preserve verification, i.e., the safety property and its dependencies define a new model that verifies successfully.

Further, Christakis et al. [16] define and detect dependencies in static code checkers. They track (potentially invalid) assumptions introduced by static code checkers and categorize assertions into verified, partially verified and not verified, depending on whether an explicit assumption was used for the verification. In comparison to our approach, they only cover explicit assumptions and, to our knowledge, do not detect transitive dependencies. Their motivation is to guide subsequent code checkers and generate unit test to verify the explicit assumptions and the not yet fully proven assertions. This is different to our motivation but could be an interesting application of dependency analysis to explore in future work.

Coverage Metrics

An application of our dependency analysis is to compute proof coverage in order to estimate the completeness of specifications and hence measure progress in verification progress. Driven by the same motivation, coverage and progress metrics have been proposed in prior work [15]. One approach is to adapt mutation-based techniques to estimate the strength of a specification by generating and verifying code mutations [27, 30]. Successfully verified mutations indicate poor specification, i.e., low coverage. IronSpec [22] instead mutates the specification. However, mutation-based coverage metrics are expensive to compute since every mutated program needs to be verified. We proposed to compute proof coverage based on the dependency graph, which only requires a single verification run to compute coverage of several methods and even individual assertions.

Alternative approaches to compute coverage in the context of model checking without mutations compute coverage based on Craig interpolation [14] or inductive validity cores [20, 21]. Further, Castaño et al. [12] determine coverage by proving safety properties encoding the condition under which statements are not covered. These approaches resemble our motivation to compute a coverage metric but they differ in the techniques used to achieve this.

Moreover, the motivation of quantifying the completeness of specifications is closely related to the need of assessing the effectiveness of a test suite in software engineering. For this purpose, several test coverage metrics, representing the fraction of code exercised while executing a test suite, and techniques to compute them have been proposed [25, 36].

Lastly, coverage metrics have been proposed to track the progress in partially verified proofs, which differs from our motivation. State space coverage [10] measures the fraction of the input space covered by a partially verified proof, i.e., for which input values the assertion was proven. GraVy [8] expresses verification progress of *partially* verified programs as the fraction of verified statements, i.e., statements that are guaranteed to not throw an exception.

Debugging Support

Our work is partly motivated by providing information to ease debugging of verification proofs. Identically motivated, ProofPlumber [13] automates assertion-based debugging by deriving and adding new assertions to a program that fails to verify. This breaks verification into smaller pieces and reveals insightful information about why verification *fails*. In contrast to that, our dependency analysis explains why an assertion *succeeds*. It might be possible to use both tools in combination as suggested in Section 8.5 (RQ5: Case Study on Analyzing the Impact of Bugs).

Further, various tools process information provided by SMT solvers, such as Z3 [18], to analyze and visualize debug information. This is comparable to exploiting UNSAT cores in our approach. For example, SMTScope [2] processes Z3 trace files and reports metrics like matching loops of quantifiers and multiplicative axioms. The motivation is to identify performance issues related to quantifiers and provide relevant debugging information. Similarly, Axolocl [23] aims at reducing the number of quantifier instantiations for performance reasons. It extracts the (quantified) axioms used to prove an assertion by analyzing the trace files and UNSAT core of Z3. Our goal is quite different since we are not interested in optimizing performance. A syntactic approach to detect dependencies was chosen by Shake [34] which over-approximates dependencies by analyzing symbols in SMT solver queries and assigning a relevance score (distance) to each assumption.

Visualizations

Program slices [32] are defined as the minimal form of a program that behaves identical to the original program. Similarly, pruning programs (see Section 5.3 (Pruning Viper Programs)) results in minimal programs that still verify with respect to their specification.

Brinkman [11] developed a tool to visualize permission flow in concurrent programs. While we have not explicitly explored this idea, the dependency graph contains permission flow information and might cover this use case as well.

Chapter 10

Conclusion

In this thesis, we detect and visualize dependencies in program verification proofs, which has several practical applications. First, it eases understanding of verification proofs by providing an explanation of why an assertion verifies successfully. Second, the impact of bugs and invalid assumptions can be estimated by detecting assertions depending on them. Lastly, verification progress can be quantified by computing proof coverage and revealing assumptions which are not covered by any proof obligation.

Dependencies are presented in the form of a dependency graph where nodes correspond to source code expressions and statements, and edges correspond to direct dependencies between them. Direct dependencies of an assertion are defined by all assumptions needed to prove it on any program path. Through transitive closure, indirect dependencies are discovered additionally, including dependencies across methods and functions.

Based on this definition, we design an algorithm to automatically detect these dependencies in Viper and Gobra programs. It builds a low-level dependency graph by exploiting UNSAT cores from SMT solvers. The dependencies detected on this lower level are lifted to Viper dependencies by keeping track of the source code expression or statement which introduced the lower level assumption or assertion. In some cases, this approach alone does not result in the desired outcome: For example, internal operations like heap summary potentially introduce several independent assumptions that should be separated from each other. To address such cases and overcome the related limitations, we implement improvement techniques which are based on customizing the information stored in nodes, such as marking them as internal thus hiding them from users, and conditionalizing conjuncts of assumptions on boolean labels to track dependencies in a more fine-grained manner.

Further, we design a command-line tool to query the resulting dependency

graphs. Apart from reporting the dependencies of assertions and the dependents of assumptions, it also computes proof coverage, which is defined as the fraction of nodes required to prove a given set of assertions.

Further, we provide an informal argument why our algorithm is sound and evaluate precision, performance, and proof coverage accuracy of the dependency analysis observed in Viper and Gobra programs. The results show that the analysis is precise in many cases and, most importantly, does not just report everything as a dependency. However, there still exist patterns causing imprecision, for example, redundant assumptions, infeasible paths, non-aliasing proofs, and quantified field assignments. The impact on verification time is non-negligible but reasonable for small and medium-sized programs. However, scalability is still a limiting factor: Verification time of a large Gobra project increased by a factor of 28 when enabling the analysis. On the other hand, queries executed on the dependency graph have a good response time, thus providing satisfiable user experience, even in large programs.

Future work could focus on the following directions:

- An important next step is to increase precision since this provides more insightful information to users and also results in more accurate proof coverage computation.
- We addressed an alternative, stronger definition for dependencies covering use cases interested in dependencies in *concrete* executions more precisely. One could explore this definition further and optimize the algorithm accordingly, for example, by making the analysis path-aware.
- One could conduct further case studies to explore how information stored in dependency graphs can be applied to various use case, for example, proof repair and assertion-based debugging.
- Another direction could be to optimize for better performance. One limiting aspect is the fact that all infeasible paths are executed. However, benchmark results suggest the existence of other major performance costs, which have yet to be found through profiling.
- In this thesis, we briefly explored how the dependency analysis can be lifted to Gobra. However, we found some unexpected results that have to be interpreted as unsoundness, we did not explore all Gobra features, and we did not attempt to improve precision of frontend encodings. These steps are important to confidently support dependency analysis for Gobra programs.
- Viper has a second backend, namely the verification condition generation backend called Carbon. We did not explore how to design a

dependency analysis algorithm for this backend but it would be an interesting extension.

- Dependency analysis should be user-friendly and accessible. Ideally, it would be directly integrated into the Viper tool chain and provide a visual user interface with several configuration options such as filtering nodes for specific properties. Further, it should visualize dependencies directly in the code editor, for instance, by highlighting source code statements and expressions. These extensions would significantly improve user experience.
- Lastly, the proposed dependency analysis algorithm could be formalized to allow for a more formal soundness argument.

Bibliography

[1] Neo4j Cypher® Manual. https://neo4j.com/docs/cypher-manual/current/introduction/. [Accessed 26-08-2025].

[2] SMTScope. https://github.com/viperproject/smt-scope. [Accessed 18-04-2025].

[3] The VerifiedSCION Project. https://www.pm.inf.ethz.ch/research/verifiedscion.html. [Accessed 20-03-2025].

[4] Gobra Benchmark Programs. https://github.com/viperproject/gobra/tree/keuscha/assumption_analysis/src/test/resources/dependencyAnalysis/benchmark_programs, 2025. [Accessed 22-09-2025].

[5] VerifiedSCION addr package. https://github.com/viperproject/VerifiedSCION/tree/master/pkg/addr, 2025. [Accessed 22-09-2025].

[6] Viper Test Programs and Benchmark Scripts. https://github.com/viperproject/silicon/tree/keuscha/assumption_analysis/src/test/resources/dependencyAnalysisTests, 2025. [Accessed 24-09-2025].

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[8] Stephan Arlt, Cindy Rubio-González, Philipp Rümmer, Martin Schäf, and Natarajan Shankar. The Gradual Verifier. In *NASA Formal Methods*, pages 313–327, Cham, 2014. Springer International Publishing.

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[10] Bernhard Beckert, Mihai Herda, Stefan Kobischke, and Mattias Ulbrich. Towards a Notion of Coverage for Incomplete Program-Correctness Proofs. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 53–63, Cham, 2018. Springer International Publishing.

[11] J. J. Brinkman. Visualizing Permission Flow of Concurrent Programs. 2016.

[12] Rodrigo Castaño, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. Verification Coverage. *CoRR*, abs/1706.03796, 2017.

[13] Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno. A Framework for Debugging Automated Program Verification Proofs via Proof Actions. In *Computer Aided Verification*, pages 348–361, Cham, 2024. Springer Nature Switzerland.

[14] Hana Chockler, Daniel Kroening, and Mitra Purandare. Coverage in interpolation-based model checking. In *Design Automation Conference*, pages 182–187, 2010.

[15] Hana Chockler, Orna Kupferman, and Moshe Vardi. Coverage metrics for formal verification. In *International Journal on Software Tools for Technology Transfer*, volume 8, pages 373–386, 2006.

[16] Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative Verification and Testing with Explicit Assumptions. In *FM 2012: Formal Methods*, pages 132–146, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[20] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient Generation of Inductive Validity Cores for Safety Properties. *CoRR*, abs/1603.04276, 2016.

[21] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats P. E. Heimdahl, and Lucas Wagner. Proof-based coverage metrics for formal verification. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 194–199, 2017.

[22] Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. IronSpec: Increasing the Reliability of Formal Specifications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 875–891, Santa Clara, CA, July 2024. USENIX Association.

[23] Kiran Gopinathan, Dionysios Spiliopoulos, Vikram Goyal, Peter Müller, Markus Püschel, and Ilya Sergey. Accelerating Automated Program Verifiers by Automatic Proof Localization. In *Computer Aided Verification*, pages 153–174, Cham, 2025. Springer Nature Switzerland.

[24] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[25] Soneya Binta Hossain and Matthew B. Dwyer. A Brief Survey on Oracle-based Test Adequacy Metrics, 2023.

[26] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[27] Alexander Knüppel, Leon Schaer, and Ina Schaefer. How much Specification is Enough? Mutation Analysis for Software Contracts. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 42–53, 2021.

[28] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016*, volume 9583, pages 41–62. Springer, 2016.

[29] Neo4j, Inc. Neo4j Graph Database. https://neo4j.com/, 2025. [Accessed 16-09-2025].

[30] Siraphob Phipathananunth. Using Mutations to Analyze Formal Specifications. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, page 81–83, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Malte H. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Doctoral thesis, ETH Zurich, Zurich, 2016.

[32] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.

[33] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification*, pages 367–379, Cham, 2021. Springer International Publishing.

[34] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn J.H. Heule, and Bryan Parno. Context Pruning for More Robust SMT-based Program Verification. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–11, 2024.

[35] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT Instability in Automated Program Verification. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 178–188, 2023.

[36] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

Appendix

Explicitly Tracking Permission Flow

In Section 4.4 (Permission Flow), we explained that dependencies introduced by permission flow are already detected by the dependency analysis but due to historic reasons we explicitly add dependencies between nodes representing the creation and removal of chunks. These explicit edges are most probably obsolete but since we did not simplify our implementation yet, we explain how these edges are added in this section. The foundational idea amounts to finding the create node corresponding to the modified or exhaled heap resource and adding dependencies accordingly.

Exhaling a Heap Resource

The simplest case concerns the removal of a heap resource, i.e., exhaling all permission to it. Such an operation only succeeds when Silicon can find a suitable chunk c with sufficient permission. The creation of this chunk c is a direct dependency of the exhale operation. Since a create node is added whenever a new chunk is created and chunks are immutable, the existence of exactly one such create node is guaranteed. In order to identify it, create nodes store a reference to the created chunk. All that is left now is adding a remove node with a dependency to the identified create node, as visualized in Figure 1.

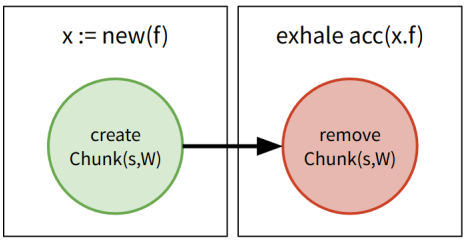

Figure 1: The visualization of the permission flow dependency for exhaling full permission of a heap resource.

Updates to Heap Resources

Heap resources can be updated by adding permission, removing partial permission, or changing the value stored in it. For the these cases, Silicon removes the existing chunk for that resource and adds a new one with updated properties. Since the new chunk is a combination of the old chunk and the update, we should add a dependency to the create node representing the old chunk. This allows us to track permissions back to the first inhale for that heap resource. When exhaling partial permission to a resource, we additionally add a remove node which also has a dependency to the old chunk's create node. A few updates to a heap resource and their dependencies are illustrated in Figure 2.

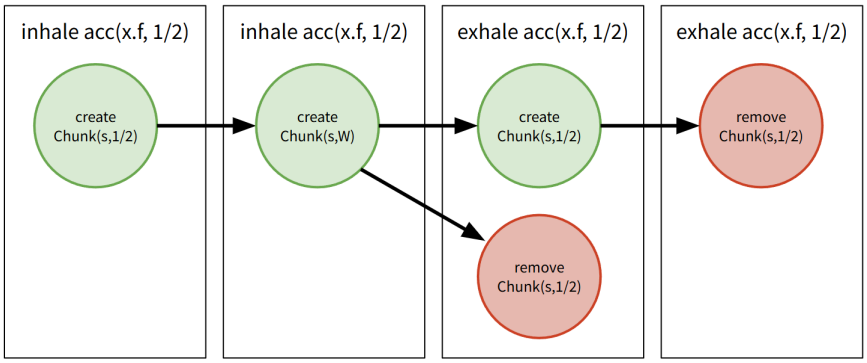

Figure 2: The visualization of the permission flow for some permission updates.

Aliasing

A special case worth mentioning is the merging of two chunks. This occurs when two heap resources are found to be aliases, for example, when explicitly assuming $x == y$ for two heap references x and y. The merge operation takes the two existing, aliasing chunks, identified by Silicon, and creates a new chunk. The new chunk is represented by a create node and has dependencies to the create nodes of both existing chunks, as depicted in Figure 3. Similarly,

non-aliasing of two heap resources might be detected, for example, when their permission would add up to more than 1. In this case, Silicon adds an assumption of the form `x != y` to the path condition. Since this assumption is a consequence of properties from the two chunks, dependencies to the corresponding create nodes are added.

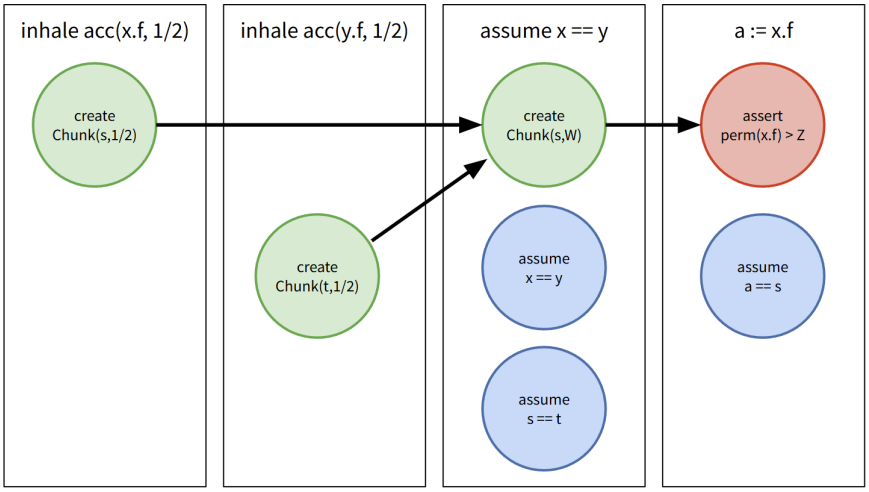

Figure 3: The visualization of the dependencies discovered when merging chunks due to aliasing.

Transferring Permissions via Method Calls

For the sake of completeness, we briefly present how calls to impure methods are handled by the dependency analysis, although this is not a special case.

Symbolically executing a method call essentially includes (1) exhaling the permissions defined by the method's precondition and (2) inhaling permission according to the postcondition. Operation (1) removes or modifies the heap resources resulting in new nodes with dependencies as discussed above. In case the method call consumes partial permission to a heap resource, operation (2) creates a new chunk with a dependency to the partial exhale caused by the precondition (see Figure 4), illustrating the frame rule. On the other hand, when the method call consumes *all* permission to a heap resource, operation (2) creates a new chunk resulting in a create node *without* any dependency, as shown in Figure 5. The connection between pre- and postcondition is then made on a higher level, namely when merging nodes as discussed in Section 4.1 (Assembling the Viper Dependency Graph). In the end, i.e., from a user's perspective, it does not make a difference which of the two applies since dependencies to the exhaled heap resources are detected regardless.

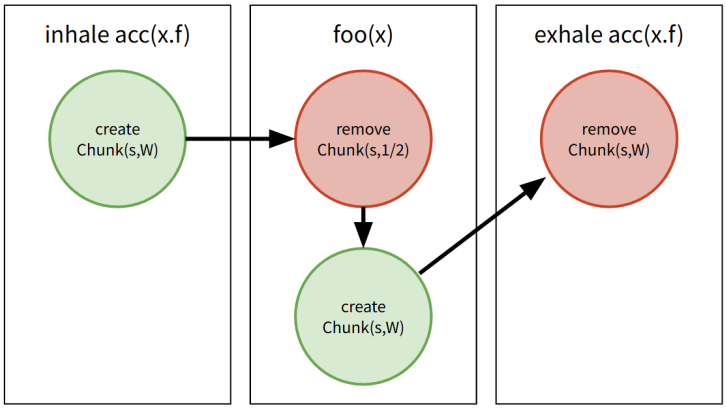

Figure 4: The permission flow dependencies for a method call that takes and returns *partial* permission to a resource. Method `foo` requires and ensures $1/2$ permission to the input `x`. Note that the method call preserves the value of `x.f`.

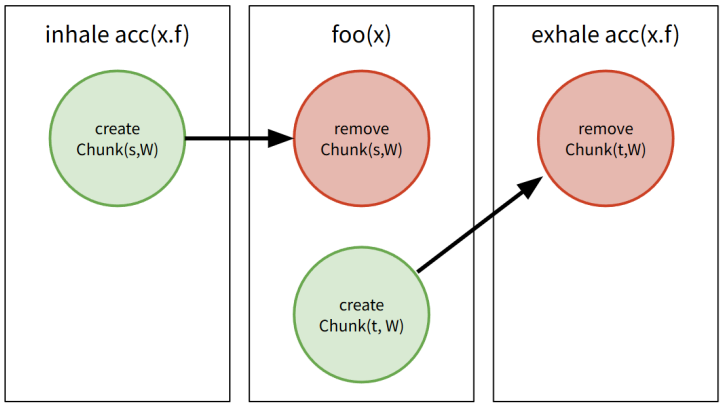

Figure 5: The permission flow dependencies for a method call that takes and returns *full* permission to a resource. Method `foo` requires and ensures *full* permission to the input `x`. Note that the method call does *not* preserve the value of `x.f`.

Predicates and Magic Wands

Predicates and magic wands are similarly encoded as exhaling and inhaling permission, thus also covered by the dependency analysis. We abstain from discussing these cases here but an example is provided Figure 6.

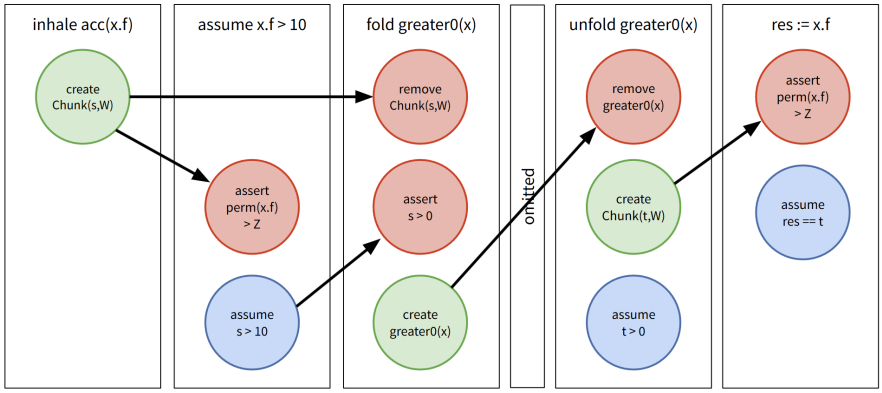

Figure 6a: The low-level dependency graph.

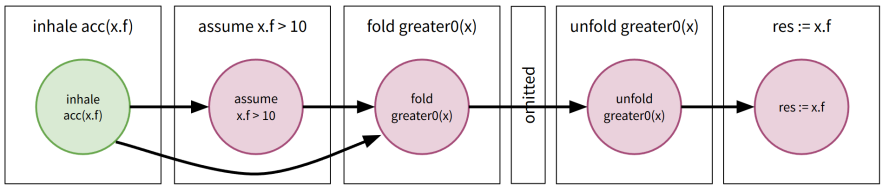

Figure 6b: The Viper dependency graph.

Figure 6: The dependency graphs of a Viper program that folds and unfolds a predicate. The predicate is defined as `predicate greater0(x: Ref){acc(x.f) && x.f > 0}`

To summarize, we always add a dependency to the latest chunk of the given resource. By transitive reasoning and the fact that chunks are immutable it follows that we keep track of the full modification history of each heap resource. In particular, we find the first inhale operation.

Binary Search Tree with a Bug

```
1  requires t.tree()
2  ensures t.tree()
3  ensures !(value in t.sortedValues())  // verifies but affected by
       bug
4  func (t *Tree) Delete(value int) {
5      unfold t.tree()
6      if (t.root != nil) {
7          t.root = t.root.delete(value, none[int], none[int])
8      }
9      fold t.tree()
10 }
11
12
13 requires lowerBound != none[int] ==> get(lowerBound) < value
14 requires upperBound != none[int] ==> get(upperBound) > value
15 requires n.tree() && n.sorted(lowerBound, upperBound)
```

```
16 ensures res != nil ==> res.tree() && res.sorted(lowerBound,
       upperBound)
17 ensures res != nil ==> !(value in res.sortedValues(lowerBound,
       upperBound)) // fails to verify
18 func (n *node) delete(value int, ghost lowerBound, upperBound
       option[int]) (res *node) {
19     unfold n.tree()
20     if (value < n.value && n.left != nil) {
21         // omitted
22     } else if (value > n.value && n.right != nil) {
23         // omitted
24     } else if (value == n.value) {
25         // delete this node
26         if (n.left != nil && n.right != nil) {
27             // find minimum in right subtree
28             // use as new value for the current node
29             // delete old node storing minimum
30             var minValue int
31             n.right, minValue = n.right.deleteMinimum(some(n.value
                   ), upperBound)
32             ghost if (n.left != nil) {
33                 n.left.convert(lowerBound, some(n.value),
                       lowerBound, some(minValue))
34             }
35             // n.value = minValue // <---- seeded bug
36             fold n.tree()
37             res = n
38         } else {
39             // omitted
40         }
41     }
42     return res
43 }
44
45 ensures t.tree() // NOT affected by bug
46 func client0(value int) (t *Tree) {
47     t = NewTree()
48     t.Insert(value)
49     t.Delete(value)
50     assert !t.pureContains(value) // verifies but affected by bug
51     return t
52 }
```

Listing 1: The relevant fragment of the Binary Search Tree program containing a bug in the delete method.

Additional Performance Results

Performance of the dependency analysis was discussed in Section 8.3 (RQ3: Performance). We provide the numeric results of the measured verification times in Table 1. They have already been visualized in Figure 8.1, page 98.

Program	#Nodes	Baseline [s]	Analysis [s]	Analysis with infeas checks [s]
Zune	46	1.31	1.16	1.23
Dutchflag	66	2.98	3.21	3.17
Relaxed Prefix	76	6.28	7.79	7.68
Example 2.1	53	2.51	2.57	2.48
Binary Tree	88	11.31	16.99	13.70
Binary Search Tree	320	15.84	23.46	16.06
VerifiedSCION addr pkg	1380	57	1585	1231

Table 1: Verification times with different Silicon configurations. Note that analysis with infeas checks corresponds to running the analysis without executing infeasible branches which is unsound.

Table 2 contains the measured response times of graph queries executed through the command-line tool. Some of these results have already been presented by Figure 8.2, page 100.

Line	#Low-Level Deps	#Gobra Deps	Mean [ms]	Std Dev
Dutchflag				
7	26	21	15.1	6.0
8	18	17	1.8	0.4
9	46	33	2.7	0.5
10	38	28	4.6	0.6
11	49	35	2.4	0.4
Example 2.1				
7	11	9	2.1	0.4
8	20	14	2.3	0.3
24	60	23	3.9	0.7
25	58	23	19.6	6.4
26	75	30	18.3	1.8
Relaxed Prefix				
12	7	6	5.4	1.6
14	35	28	9.0	1.0
15	31	24	6.3	1.1
16	35	26	5.8	8.5
29	6	6	2.4	0.5
31	12	11	2.1	0.3
36	30	23	6.8	1.0
37	14	13	2.5	0.6
Zune				
36	8	8	1.1	0.3
37	10	9	0.8	0.3
38	2	2	0.6	0.1
58	3	3	0.5	0.1
Binary Search Tree				
302	51	16	12.3	1.3
304	370	70	838.8	80.0
306	673	160	1952.5	48.5
167	30	10	11.6	1.5
VerifiedSCION addr package				
isdas@113	90	25	209.9	14.0
isdas@80	283	58	318.4	18.2
host@478	188	32	340.2	13.8
host@269	12	6	207.2	14.5
host@270	17	7	204.5	10.3
fmt@144	41	21	206.8	12.7

Table 2: Raw results of the benchmarks measuring query response times in the command-line tool.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

◉ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Dependency Analysis in Automated Verification Proofs

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Keusch	Andrea

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zurich, 01.10.2025	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard