ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Recovering From Verification Failures

Bachelor Thesis

Josua Stuck

September 5, 2021

Advisors: Prof. Dr. P. Müller, Dr. M. Schwerhoff
Department of Computer Science, ETH Zürich

Abstract

Silicon is a verification backend for the permission-based Viper verification language. Programs encoded in Viper contain various specifications that Silicon verifies using symbolic execution; a way to execute a program with symbolic rather than concrete values.

The current version of Silicon stops the verification after a single failure. In this thesis we designed and, in part implemented, ways for Silicon to recover from different types of failed assertions, and continue the verification.

The changes presented in this thesis increase the number of errors reported by a single run of Silicon, while not substantially increasing verification times.

Contents

Chapter 1

Introduction

Viper [7] is a verification language and a suite of tools developed at ETH Zurich. Viper comprises an intermediate verification language as well as two verification backends. Figure 1.1 gives an overview of the Viper infrastructure. The Viper language supports encoding various program specifications. For example, pre- and postconditions for methods or invariants for loops can be encoded. Viper programs can either be written directly in Viper or translated from another high-level language by one of the various frontends. The correctness of those programs can then be verified by either of the two backends: Silicon [8] or Carbon [6].

Silicon, which is based on symbolic execution, stops the verification as soon as the first error is found. The goal of this thesis is to enable Silicon to recover from failed assertions , that is, to report an error and continue the verification. If an assertion fails, it would be unsound to just continue the verification, as the point of an assertion is to only continue a program if it holds. In order to soundly recover from a verification failure, Silicon's state needs to be adapted. That is, Silicon needs to be in a state in which the failed assertion would have held. The changes to the state that need to be performed depend on the type of the failed assertion; Section 2.1.3 explains the difference between pure and impure assertions.

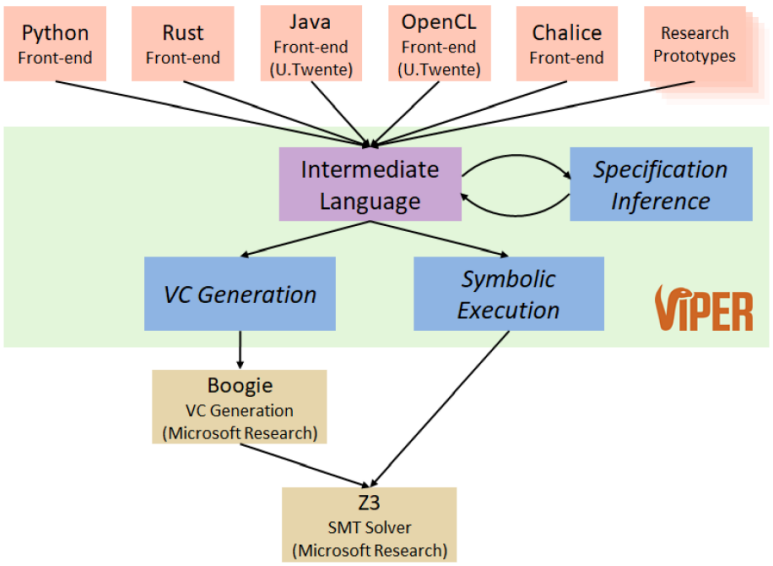

Figure 1.1: The Viper intermediate language with its accompanying tools, as shown on [4]. Viper programs that are either written directly or generated by one of the frontends can be verified by either the Verification Condition Generation backend or the Symbolic Execution backend. Both backends ultimately make use of Z3 — the underlying SMT solver.

1.1 Chapter Overview

- In Chapter 2, the necessary background information about Viper and Silicon is introduced.
- In Chapter 3, Silicon's branching behavior is discussed, and a way to continue the verification after an error on one path as well as its implementation is presented.
- In Chapter 4, a way to recover from failed pure assertions is presented.
- In Chapter 5, a way to recover from failed impure assertions is proposed.
- In Chapter 6, the implications on performance of the changes proposed in this thesis are evaluated.
- In Chapters 7 and 8 conclusions are drawn and possible future work is discussed.

Chapter 2

Background

2.1 Viper

Viper is a permission-based intermediate verification language as well as a collection of software verification tools developed at ETH Zurich by the Programming Methodology Group. There are several Viper frontends for high-level languages such as Python or Rust, as well as two different verification backends.

2.1.1 Overview

Listing 2.1 shows the Viper features that are relevant for this thesis. A more complete overview of the Viper language can be found here [3].

```
1  field f: Int
2
3  function sqr(x: Ref): Int
4  requires acc(x.f, 1/3)
5  {
6      x.f * x.f
7  }
8
9  method viper(x: Ref)
10 requires acc(x.f, 1/2)
11 {
12     assert sqr(x) >= 0
13     exhale acc(x.f, 1/2)
14 }
```

Listing 2.1: An example of a program written in the Viper language.

On line 1 of Listing 3.1, a *field* of type `Int` is declared. Every object (accessed via a `Ref`) has all fields that are declared in a program. On line 3 a *function*,

which takes a reference to an object and returns an integer, is declared. Functions are side-effect free, that is, they don't modify the program state. The function's *precondition* is stated on line 4, and specifies that the function needs read access to `x`'s field `f`. The meaning of `acc(...)` is further explained in Section 2.1.2. Starting from line 9 a *method* is declared. Methods can, in contrast to functions, alter the program state. The `assert` statement on line 12 checks that the result of the application of function `sqr` is nonnegative. Finally, the `exhale` statement on line 13 gives away the permissions that were gained from the method's precondition. Alternatively, this could have been encoded as a postcondition `ensures acc(x.f, 1/2)`.

2.1.2 Permissions

A central part of the Viper language is the *accessibility predicate*, which is used to denote permissions to memory locations. For arbitrary well-typed expressions e_1, e_2, a predicate of the shape `acc(`e_1`.f, `e_2`)` denotes e_2 permissions to the memory location identified by $e_1.f$. Permission amounts can be fractions, that is, permission amounts greater than 0 (or *none*) grant read access, and a permission amount of 1 (or *write*) means write access.

2.1.3 Assertions

Assertions in Viper are either *pure* or *impure*. Impure assertions are assertions that add or remove permissions from the program state. All other assertions are pure. In Viper, assertions can be *inhaled*, *exhaled*, *asserted*, or *assumed*. Inhaling an assertion entails gaining all permissions that are included in the assertion, as well as assuming all pure sub-assertions. On the other hand, exhaling an assertions means giving away the permissions specified in the assertion, as well as checking the pure sub-assertions. Asserting an assertion exhales the assertion, but continues the verification in the pre-exhale state. Assuming an assertion works the same as inhaling an assertion, but the assertion can only be pure[1].

The impure assertion `acc(x.f) && x.f > 0` is a conjunction of the impure sub-assertion `acc(x.f)` and the pure sub-assertion `x.f > 0`. Inhaling this assertion adds write permissions to `x.f` to the program state, and assumes that $x.f > 0$. Exhaling the assertion tries to remove write permissions to `x.f` from the state and, if successful, checks if $x.f > 0$.

In Listing 2.1, the pure assertion `sqr(x) >= 0` is asserted on line 12, whereas the impure assertion `acc(x.f, 1/2)` is exhaled on line 13.

[1] [2] enabled assuming impure assertions, but it is not enabled by default in Silicon.

2.2 Silicon

Silicon is one of two verification backends for Viper, and is based on symbolic execution. In the rest of this chapter we will introduce Silicon by taking an excerpt from [8], where a more in-depth explanation can be found.

2.2.1 Symbolic Execution

The general idea of symbolic execution [5] is to execute a program using a *symbolic state* that contains symbolic rather than concrete values. Each executed statement updates the symbolic state according to its (abstract) semantics. A method implementation can be verified by setting up a symbolic state such that the method's precondition holds, symbolically executing the method, and then checking whether the postcondition holds (in the symbolic state resulting from the symbolic execution). Silicon uses Smallfoot-style symbolic execution, named after Smallfoot [1] – the first automated verifier for separation logic. Such verifiers separate their symbolic state into *path conditions* and a *symbolic heap*. The path conditions correspond to assertions expressed in first-order logic, for example, equalities between symbolic values, whereas the symbolic heap corresponds to assertions that cannot be expressed directly in first-order logic, such as permissions to fields or predicate instances.

2.2.2 Initial Definitions

In order to understand how Silicon works, some initial definitions are needed:

Definition 2.1 Let R be the type of *verification results*, that can be either `success` or `failure`. Additionally, `failure` contains some message about the reason or type of the failure.

Definition 2.2 Let Σ be the type of *symbolic states*, with σ denoting a single symbolic state. The entries contained in a symbolic state that are relevant for this thesis are the following:

- A *store* γ of type Γ, a map from local variables to their symbolic values.
- A set of*path conditions* π of type Π. The set comprises all assumptions that have been made over the course of the verification.
- A *symbolic heap* h of type H. The heap contains information about which memory locations are currently accessible, as well as those locations' symbolic values.

Definition 2.3 Let S be the type of statements, with typical element s; let A be the type of assertions, with typical element a; let E be the type of expressions, with typical element e; let V be the type of symbolic values, with typical element v.

Definition 2.4 Let $fresh$ be used to denote a fresh symbolic value, that is, a value which we know nothing about (it could have any value).

Definition 2.5 Let the operator $ite(v_1, v_2, v_3)$ denote a function that returns v_2 if v_1 is `true`, and v_3 otherwise.

Symbolic Execution Primitives

Silicon's symbolic execution is based on four symbolic execution primitives. The last argument passed to those primitives is a *continuation*. The continuation is a function that is invoked with the updated state and that contains the remainder of the verification that still needs to be performed. Q typically denotes a continuation.

- exec: $\Sigma \to S \to (\Sigma \to R) \to R$
  is used to execute a statement in a given state.
- produce: $\Sigma \to A \to Snap^2 \to (\Sigma \to R) \to R$
  is used to inhale assertions.
- consume: $\Sigma \to A \to (\Sigma \to Snap \to R) \to R$
  is used to exhale assertions.
- eval: $\Sigma \to E \to (\Sigma \to V \to R) \to R$
  is used is used to symbolically evaluate expressions in a given state Σ. The symbolic value obtained by evaluating an expression e is typically denoted by e'.

2.2.3 Symbolic Heap

Symbolic heaps are multisets of *heap chunks*. The kind of heap chunks relevant for this thesis are *field chunks*. Other types of heap chunks exist, see [8].

Definition 2.6 Let $id(r; v, p)$ denote a *field chunk* with the field receiver r, field name id, symbolic value v, permission amount p.

Definition 2.7 Let $id(r)$ denote a *field chunk identifier*, used for finding chunks that match a field access.

Definition 2.8 Let $perm(ch)$ denote the permission amount held by a field chunk ch, and let Z denote no permission.

2.2.4 State Consolidation

There are two main reasons why Silicon needs to perform *state consolidations* from time to time. First, Silicon separates the symbolic state into a set of path

[2]Snaps aren't relevant until Chapter 5, and will be explained in Section 5.3.

conditions and a set of heap chunks. Some information can be implicitly available from the heap, but is not explicitly stated as a path condition. For example, if there are two heap chunks with a combined permissions amount that is greater than 1, it is implied that the chunk's receivers cannot be aliases. Secondly, there is an asymmetry between adding and removing permissions from the state. While Silicon creates a new heap chunk whenever it adds permissions to the state, when removing permissions, it looks for a single heap chunk with the necessary permissions[3]. This means that there are situations where there are enough permissions available, but they cannot be removed because no single heap chunk can be found with sufficient permissions.

The goal of a state consolidation is to minimize the incompletenesses that can arise from the two points discussed above. To solve the first point, the state consolidation algorithm adds path conditions to the state that encode which references cannot possibly be aliases. To solve the second point, the algorithm merges heap chunks that definitely belong to the same memory location into a single chunk. State consolidations will be relevant when discussing a problem that arises in Chapter 4.

[3]There is an alternate non-greedy method for removing permissions. It is however, not enabled by default.

Chapter 3

Reporting One Error per Branch

The goal of this thesis ultimately was to enable Silicon to report more errors. In this chapter we describe how we changed Silicon's branching behavior to allow for more errors to be reported.

3.1 Enabling Silicon to Report One Error per Branch

Consider the following Viper program:

```
1 method branching(b:Bool){
2     if(b){
3         assert b
4     } else {
5         assert b  //error not reported
6     }
7     assert false //error reported
8 }
```

Listing 3.1: Example of how branching over conditionals in Silicon leads to potentially confusing error reporting. The assertion on line 5 seemingly verifies successfully.

One can easily see that the assertion on line 5 of Listing 3.1 should not verify. However, the current version of Silicon does not report an error on line 5. Instead, the `assert false` on line 7 generates a verification failure. A user who is not familiar with Silicon's branching behavior might think that every statement up until line 7 successfully verified, but that is not the case. Silicon branches over conditionals such as `if-then-else` statements. That is, it executes the `then` branch (as well as the rest of the program) under the assumption that the condition is `true`, and the `else` branch under the assumption that the condition is `false`. In Listing 3.1, the branch where `b` is true is verified first. On line 7 the `assert false` statement leads to a

verification error, resulting in the path where `b` is false to never be executed. Hence the error on line 5 is missed.

3.1.1 Branching in Silicon

Silicon's combine operator shown in Figure 3.1 can be used to compose verification results. It takes as arguments two operations that take no arguments but return verification results of type R. To emphasize that these operations are not verification results yet, but need to be evaluated first, we use the notation $() \to R$. Notice how Q_1 is evaluated in every case, but Q_2 is only evaluated if Q_1 doesn't evaluate to a failure.

```
1: ⊕ : (() → R) → (() → R) → R
2: Q1 ⊕ Q2 =
3:     r1 := evaluation of Q1
4:     if r1 is a failure then
5:         r1
6:     else
7:         evaluation of Q2
```

Figure 3.1 Silicon's combine operator, that can be used to obtain a single verification result from two results.

Figure 3.2 shows a slightly simplified version of Silicon's branching operation. It takes as arguments a state, a symbolic value which is the conditionals that is branched over, as well as two continuations. The continuations themselves take as an argument a state, under which they are evaluated. Note that this means that if the result of Q_v is `failure`, $Q_{\neg v}$ is never evaluated, that is, the `else` branch is never symbolically executed.

```
1: branch: Σ → V → (Σ → R) → (Σ → R) → R
2: branch(σ, v, Q_v, Q_¬v) =
3:     Q_v (σ{π := σ.π ⋃ v})
4:     ⊕
5:     Q_¬v (σ{π := σ.π ⋃ ¬v})
```

Figure 3.2 Silicon's branching rule, that is used to branch over conditionals. One path is evaluated under the assumption that the conditional is true, the other path under the assumption that the conditional is false.

Figure 3.3 shows an updated version of the combine operator. We added a field `previous` to the type of verification results, which can be used to record other results that occurred during the verification. This allows us to evaluate both branches even if the first branch results in a failure and keep both results. What might not be immediately clear is why r_2 is not always

```
1: Q1 ⊕ Q2 =
2:     r1 := evaluation of Q1
3:     r2 := evaluation of Q2
4:     if r1 is a failure then
5:         r1.previous := r2 ∪ r2.previous
6:         r1
7:     else
8:         r2.previous := r1 ∪ r1.previous
9:         r2
```

Figure 3.3 The updated combine operator, which can be used to obtain a single verification result from two results. If either of the results is a failure, the returned result will be a failure as well.

appended to r_1 but the other way around if r_1 is not a failure. The intention behind this is that if either result is a failure, then the result returned should also be a failure. This allows Silicon to correctly determine the overall result of the branching operation.

3.2 Reporting Errors That Appear on Multiple Branches

With the changes introduced in Section 3.1 the second path resulting from branching over a conditional is verified even if the first path contains an error. This means that an error can either occur on just one of the branches or on both. For the user, it would be helpful to know if an error occurred on multiple or just one branch. Or more specific: under which branch conditions an error occurred. In this section, we explain how we adapted Silicon's error reporting to provide more useful information to the user.

```
1 method branching2(b: Bool, x: Int){
2     var y: Int
3     if (b){
4         y := x
5     } else {
6         y := x*x
7     }
8     assert b && y >= 0 //Assertions fails on two branches
9 }
```

Listing 3.2: Example of an error that occurs on multiple branches.

The assertion on line 8 of Listing 3.2 fails to verify two times: once on the path where `b` is true (but `y` could be negative) and once on the path where `b` is false.

We changed Silicon such that whenever a failure is created, the current branch conditions are recorded alongside the failure. Every failure has a so called

failure context which, among other information about the failure, contains the branch conditions under which the failure happened. When the errors encountered during the verification are reported, Silicon reports under which branch conditions they ocurred. For the assertion in this example, Silicon reports that `b` failed under branch conditions `!b`, while `y >= 0` failed under branch conditions `b`. This can help the user better understand why the error happened.

```
1 Assert might fail. Assertion b might not hold. (L8)
2          under branch conditions:
3          !b [ L3 ]
4 Assert might fail. Assertion y >= 0 might not hold. (L8)
5          under branch conditions:
6          b [ L3 ]
```

Listing 3.3: Silicon's output when verifying Listing 3.2. The location information (file and line number) have been trimmed.

Chapter 4

Recovering From Failed Pure Assertions

In this chapter we describe how we enabled Silicon to recover from failed *pure* assertions. Recovering from failed impure assertions is more challenging and will be discussed in Chapter 5. We will present an updated consume rule.

4.1 Background

Recall from Section 2.1.3 that pure assertions do not modify the heap. From Section 2.2.2, recall that both asserting and exhaling assertions in Silicon is done by consuming the assertion. The difference between the two is that when exhaling, the verifier continues in the post-consume state, but when asserting an assertion it continues in the pre-consume state. Thus, to recover from failed pure assertions we needed to change the consume rule.

Consider the following Viper program:

```
1 method pure(xs: Seq[Int], i: Int){
2     assert i >= 0
3     assert i < |xs|
4
5     var e: Int := xs[i]
6
7     assert e == 42
8 }
```

Listing 4.1: A Viper program with failing pure assertions. The original Silicon version stops the verification when the first assertion fails.

Verifying the method shown in Listing 4.1, the current version of Silicon reports an error on line 2, and stops the verification. At the beginning of the

method, nothing is known about the argument `i`, so neither of the assertions on line 2 and 3 would hold. When accessing an element of a sequence, Silicon checks if the provided index is guaranteed to be within bounds, and reports an error otherwise. Without the first two assertions, the assignment on line 5 would fail, because `i` is not guaranteed to be a valid index. If the first two assertions successfully verify however, the index is guaranteed to be within bounds and the assignment doesn't fail.

Pure assertions can only be expressions, conjunctions of pure sub-assertions, or (pure) conditionals. Consuming a conjunction of pure sub-assertions is done by consuming each sub-assertion individually. Consuming pure conditionals of the shape $e\,?\,a_1 : a_2$ is done by branching over the conditional e, and then, depending on the branch, consuming a_1 or a_2. This means that recovering from failed pure assertions boils down to recovering from the failed consumption of an expression.

Figure 4.1 shows Silicon's consume rule for expressions. The expression is evaluated to its symbolic value, then the underlying SMT solver is queried to check if the expression's symbolic value is true in the given state. Only if the assertion is true the continuation Q is invoked.

```
1: consume(σ1, e, Q) =
2:     eval(σ1, e, (λσ2, e′ ·
3:         if σ2 ⊨ e′ then
4:             Q(σ2, σ2.h, unit)
5:     ))
```

Figure 4.1 Silicon's consume rule for pure assertions.

4.2 Recovering From a Failed Pure Assertion

Before explaining how such a failed pure assertion can be recovered from, we need to explain what exactly is meant by recovering. The goal is to report the error, but continue the verification in a state where the error would not have happened (i.e. the assertion would have held), to find and report further verification errors. This can be done by adding, as path constraints, that the assertion held. Adding path constraints can be seen as eliminating execution paths: by adding a path constraint that states that the assertion held, we eliminate all execution paths in which it couldn't hold.

Figure 4.2 shows the updated consume rule for pure assertions. If the assertion is not true, we add it as a path constraint, create an error, and continue the verification. The result of this change is that the verification doesn't stop after the assertion on line 2 of Listing 4.1 generates an error. Instead, the error is recovered from and the assertion on line 3 is verified,

also generating an error. After recovering from those assertions, the path conditions contain that $0 \leq i < |xs|$, and the assignment on line 5 passes the index bound checks. Finally, the assert statement on line 7 generates an error.

Note that we only add path constraints, not change them. If for example, after line 3 we added an assertion `assert i < 0`, and recovered from the failed assertion, then the path constraints would contain both `i >= 0` and `i < 0`. Those constraints together would be unsatisfiable, the verifier would end up in an infeasible path, and the verification would stop.

```
1: consume(σ1, e, Q) =
2:     eval(σ1, e, (λσ2, e′ ·
3:         if σ2 ⊨ e′ then
4:             Q(σ2, h, unit)
5:         else
6:             failure() ⊕ Q(σ2{π := σ2.π ⋃ e′}, h, unit)
7:     ))
```

Figure 4.2 Silicon's updated consume rule for pure assertions that supports recovering from failed assertions. When the failure is created, the current branch conditions are recorded.

4.2.1 Silicon's Try Mechanism

The updated consume rule in Figure 4.2 still has some drawbacks, and cannot be used as-is in practice. The reason for this is the way Silicon performs state consolidations. Recall that state consolidations, introduced in Section 2.2.4, are needed to overcome heap-related incompletenesses. Because state consolidations are expensive operations (worst-case complexity is cubic in the number of heap chunks), they are not performed after every state update. Figure 4.3 shows Silicon's *try mechanism*. The `try` operation takes as arguments a state and two continuations, the *action continuation* Q_{action} and the usual remainder of the verification Q. Q_{action} itself takes a continuation as an argument. If Q_{action} (e.g. the consumption of an assertion) fails, a state consolidation is performed and the action is executed again. When Q_{action} is evaluated it is passed a continuation that sets res_{local} to success. In the original Silicon version, reaching this statement on line 7 meant that Q_{action} verified successfully (because its continuation was invoked). In Silicon, consume operations are wrapped inside a try block.

From now on, we'll refer to the operation that actually consumes an assertion (as shown in Figures 4.1 and 4.2) as *consume′*, and to the wrapped operation as *consume*. Figure 4.3 shows how the consumption is wrapped in a try block.

```
1:  try: Σ → (Σ → (Σ → R) → R) → (Σ → R) → R
2:  try(σ, Q_action, Q) =
3:      res_local := failure()
4:      res_global :=
5:          Q_action(σ,
6:              (λ σ_1 ·
7:                  res_local := success()
8:                  Q(σ_1)))
9:      if res_local is a failure then Q_action (σ{(h, π) := consolidate(σ.h, σ.π)}, Q)
10:     else res_global
11:
12: consume(σ, a, Q) =
13:     try(σ, (λ σ_1, Q_succ · consume'(σ_1, a, Q_succ)), Q)
```

Figure 4.3 Silicon's original try function that triggers a state consolidation when Q_{action} fails.

```
1 field f: Int
2 method retry(x: Ref, y: Ref){
3     inhale acc(x.f) && acc(y.f)
4     assert x != y
5 }
```

Listing 4.2: Example of a pure assertion that fails before a state consolidation, but passes afterwards.

Listing 4.2 is an example that illustrates how Silicon's retry behavior combined with the consume rule from Figure 4.2 can lead to incompleteness. From the inhale statement on line 3, we know that `x` and `y` can't be aliases (their permission amounts to field `f` add up to more than 1). Thus, the assert statement on line 4 should verify. The verifier's state however, reflects this disequality only *after* a state consolidation has been performed. When `assert x != y` is consumed, Silicon erroneously reports an error. The reason for this behavior is that when the consumption fails, Silicon recovers from the failure, reports an error and subsequently invokes the continuation (i.e. continues with the verification), instead of performing a state consolidation an retrying the consumption. Invoking the continuation passed to *consume'* by the try operation sets res_{local} to `success()` (line 7 of Fig. 4.3), and thus skips the state consolidation and retrying of the consumption.

To solve this problem, we need to make sure that failures are not recovered from if they occur in a try block before the state consolidation. In order to reliably determine whether the verification currently is inside a try block and before the state consolidation, we added a counter *retryLevel* to the state that keeps track of how many (possibly nested) try blocks have been entered (and have not yet performed a state consolidation). A *retryLevel* of 0 means

that either the verifier is not in a try block, or that every try block that has been entered but not yet exited has already performed its state consolidation. Figure 4.4 and 4.5 show the final consume rule for pure assertions and the updated try operation, respectively.

```
1:  consume'(σ1, h, e, Q) =
2:      eval(σ1, e, (λσ2, e' ·
3:          if σ2 ⊨ e' then
4:              Q(σ2, h, unit)
5:          else
6:              if σ2.retryLevel == 0 then
7:                  σ2.π := σ2.π ⋃ e'
8:                  failure() ⊕ Q(σ2, h, unit)
9:              else
10:                 failure()
11:     ))
```

Figure 4.4 Silicon's final updated consume rule for pure assertions that supports recovering from failed pure assertions. Failures are only recovered from if the verifier is either not in a try block or each entered but not exited try block's state consolidation has been performed.

```
1:  try(σ, Q_action, Q) =
2:      rl := σ.retryLevel
3:      res_local := failure()
4:      res_global :=
5:          Q_action(σ{retryLevel := rl + 1},
6:              (λ σ1 ·
7:                  res_local := success()
8:                  Q(σ1)))
9:      if res_local is a failure then
10:         Q_action(σ{h, π := consolidate(σ.h, σ.π), retryLevel := rl}, Q)
11:     else res_global
```

Figure 4.5 Silicon's updated try function that modifies the state to keep track if the state has been consolidated. The try function is identical to the try function in Figure 4.3 except for the handling of the counter *retryLevel*.

4.3 Limiting the Number of Errors Reported

With the changes from Chapter 3 and Chapter 4, Silicon verifies all paths resulting from branching over conditionals and recovers from failed pure assertions. This means that Silicon reports more than just one error per

program. However, a user might want to only run the verification until a certain number of errors are found, either for performance or simplicity reasons. To enable specifying the number of errors that should be reported before the verification is stopped, two changes were necessary: First we added a counter to the Verifier that is increased every time a `failure` is created. Second, we changed the behavior of the `combine` operator, such that it only evaluates the second argument if the number of errors that should be reported has not yet been reached. Figure 4.6 shows the updated `combine` operator.

```
1:  Q1 ⊕ Q2 =
2:      r1 := evaluation of Q1
3:      if more errors should be reported then
4:          r2 := evaluation of Q2
5:          if r1 is a failure then
6:              r1.previous := r2 ∪ r2.previous
7:              r1
8:          else
9:              r2.previous := r1 ∪ r1.previous
10:             r2
11:     else
12:         r1
```

Figure 4.6 Updated combine operator that only continues the verification (by evaluating the second argument) if more errors should be reported. Lines 4 to 10 are identical to lines 2 to 8 of Figure 3.3.

Chapter 5

Recovering From Failed Impure Assertions

In this chapter we will describe how Silicon could be enabled to recover from failed impure assertions. We will present the formalization of an updated `consume` rule that was not implemented.

5.1 Background

Recall from Section 2.1.3 that impure assertions are assertions that manipulate the heap, that is they add or remove permissions. As with recovering from pure assertions, the consume rule will need to be changed as well.

Consider the following Viper program:

```
1 field f: Int
2 method impure(x: Ref, y: Ref, z: Ref){
3     inhale acc(x.f, 1/3)
4     inhale acc(y.f, 1/6)
5     inhale acc(z.f, 1/3)
6
7     exhale acc(x.f, 1/2)
8 }
```

Listing 5.1: An example of a failing, but recoverable, exhale statement (line 7)

Inhaling the accessibility predicates on lines 3 to 5 of Listing 5.1 adds three field chunks to the heap. Field chunks are a type of heap chunk, and specify a field receiver, a field name, the location's symbolic value, and a permission amount. Inhaling `acc(x.f, 1/3)` adds a field chunk with receiver `x`, field name `f`, a fresh symbolic value `v`, and a permission amount of 1/3. If we want to represent a heap containing such a field chunk, we write the following: `x.f -> v # 1/3`.

The verification of the `exhale acc(x.f, 1/2)` statement of Listing 5.1 will produce an error. Silicon tries to find a field chunk for `x.f` with a permission amount of at least 1/2, to remove 1/2 permissions from this chunk. However, the heap only contains a chunk with 1/3 permissions to `x.f`. Therefore, the consumption of `acc(x.f, 1/2)` fails.

5.2 Recovering From a Failed Impure Assertion

Recall how we enabled Silicon to recover from failed pure assertions by eliminating all traces in which the assertion doesn't hold, by adding the assertion as a path constraint. For impure assertions this is not sufficient, as the consumption of such assertions removes permissions from the state. The general idea however stays the same: We need to change Silicon's state to a state in which the assertion would have held. In the case of accessibility predicates, this means bringing Silicon into a state where enough permissions are available to consume the predicate, and then removing the permission amount specified by the predicate.

But where can we soundly take additional permissions from? In Viper, references can point to any object, and multiple references can point to the same object. References that point to the same object are called aliases. Looking at the example in Listing 5.1 we can make an important observation: If `x` and `y` were aliases, their combined permissions would add up to 1/2. Similarly, if `x` and `z` were aliases, their permissions would add up to 2/3. This means that states where `x` aliases with at least one of `y` or `z`, contain enough permissions to consume the accessibility predicate from line 7.

Conceptually, we need to do two things in order to recover from the failed `exhale` in Listing 5.1. First, we need to filter out all remaining program traces in which not enough permissions to `x.f` are available. Secondly, we need to remove 1/2 permissions from the resulting state.

In practice it is sufficient to carefully update the relevant field chunks' permission values and record in the path constraints how they relate to the old permission values. Aliasing between references is then implied by those constraints.

It is important to understand the relationship between the permission values before and after an exhale statement. The permission amount actually available for (the memory location denoted by) `x.f` can be expressed as the sum $p_{before} = 1/3 + ite(x = y, 1/6, Z) + ite(x = z, 1/3, Z)$. This sum conditionally adds up the permission values of possible aliases of `x`. The statement `perm(x.f)` returns such a summary of the available permissions. After the exhale statement, this sum needs to be 1/2 less. If p_{after} denotes the permission sum for `x.f` after the consume operation, then the following

equation has to hold:

$$0 \leq p_{after} = p_{before} - 1/2 \tag{5.1}$$

Because the underlying SMT solver operates on real numbers, but permissions cannot be negative, the fact that $0 \leq p_{after}$ has to be explicitly assumed. As promised, Equation 5.1 implies that `x` has at least one alias. Equation 5.2 shows Equation 5.1 with p_{before} expanded.

$$0 \leq p_{after} = 1/3 + ite(x = y,\ 1/6,\ Z) + ite(x = z,\ 1/3,\ Z) - 1/2 \tag{5.2}$$

If neither `y` nor `z` are aliases of `x`, Equation 5.2 can be simplified to $0 \leq 1/3 - 1/2$, which clearly doesn't hold. Therefore at least `y` or `z` must be aliases of `x`.

What remains is the construction of p_{after}. Because it is a sum representing the permission amounts available for `x.f` after the consume operation, it conditionally sums up the post-consume permission values of all field chunks for field `f`. Due to all the possible aliasing combinations, directly creating terms for the new permission values is not feasible. Instead, we replace the field chunks' permissions with fresh symbolic values. Fresh symbolic values are symbolic values which don't have any constrains imposed on (yet). This means that by assuming Equation 5.1, we constrain the fresh permission values to satisfy the semantics of the consume operation.

5.3 Updated Consume Rule

Figure 5.1 shows an updated version of Silicon's `consume'` rule. When consuming an accessibility predicate `acc(x.f, p)` the algorithm loops over all field chunks for field `f`, and iteratively sums up p_{before} and p_{after}.

The expression $perm_{available}$ represents the permissions provided by the loop's current field chunk, depending on whether the chunk's receiver is an alias of the accessibility predicate's receiver. Whether the receivers are aliases isn't statically known at this point, and the expression needs to be evaluated by the underlying SMT solver. By summing up each chunk's $perm_{available}$ we construct p_{before}. On line 14, the expression p_{taken} is constructed: An expression that also needs to be evaluated by the SMT solver, and returns the min of $perm_{available}$ the amount of permissions still needed. We need p_{taken} to construct p_{needed} on line 18, which is then used to decide if enough permissions were available *without* taking any permissions from possible aliases.

On line 16, the current chunk's permission value is replaced by a fresh symbolic value. After the loop, on line 21, Equation 5.1 is assumed by adding it as a path constraint. Checking if sufficient permissions were available without taking any permissions from possible aliases happens on line 23, to decide whether a verification failure needs to be returned along with the result of the continuation.

```
1: consume': Σ → A → (Σ → Snap → R) → R
2: consume'(σ1, acc(id(r), p), Q) =
3:     eval(σ1, p :: r, (λσ2, p' :: r' ·
4:         Let hr ⊆ σ2.h contain all heap chunks for identifier id
5:         ho := σ2.h \ hr
6:         pneeded := p'
7:         hn := ∅
8:         pbefore := Z
9:         pafter := Z
10:        s := fresh
11:        foreach id(r2; v2, p2) ∈ hr do:
12:            permavailable := ite(r2 == r', p2, Z)
13:            pbefore += permavailable
14:            ptaken := min(permavailable, pneeded)
15:            pfresh := fresh
16:            hn := hn ⋃ id(r2; v2, pfresh)
17:            pafter += ite(r2 == r', pfresh, Z)
18:            pneeded := pneeded − ptaken
19:            σ2.π := σ2.π ⋃ (r2 == r' ∧ pfresh > Z ⇒ s == v2)
20:        done
21:        σ2.π := σ2.π ⋃ (pafter == pbefore − p') ⋃ (0 ≤ pafter)
22:        h := hn ⋃ ho
23:        if pneeded == Z then
24:            Q(σ2, s, h)
25:        else
26:            failure() ⊕ Q(σ2, s, h)
27:    ))
```

Figure 5.1 Silicon's updated consume rule for impure assertions, that supports recovering from failures by taking possible aliases into account.

The consume operation passes to its continuation a snapshot. In the case of exhaling an accessibility predicate, snapshots represent the symbolic value of the memory location denoted by the predicate. That is, the snapshot returned when consuming an accessibility predicate is equal to the symbolic value of the memory location specified in the predicate. All field chunks with nonzero permissions whose receivers alias with the receiver of the

accessibility predicate that is consumed must contain the same symbolic value, which must be equal to the snapshot returned. On line 10 the snapshot is assigned a fresh symbolic value. On line 19, the relationship between the snapshot and the current field chunk's symbolic value is assumed.

The algorithm shown in Figure 5.1 is based on the *consumeComplete* consumption algorithm presented in [9], which is again based on the algorithm for consuming quantified permissions presented in [8]. All three algorithms take a global view on the heap (as opposed to the greedy consumption algorithm), and construct conditional sums of available permissions.

Chapter 6

Evaluation

In this Chapter we will discuss the implications on performance introduced by the changes made to Silicon in the scope of this thesis. We benchmarked the (at the beginning of the thesis) current version against the final version resulting from this thesis. The average verification times slightly improved.

As the baseline for our performance comparison, Silicon version with commit hash `6fa45645` (the current version at the start of this thesis) was used, and compared to the final version of this thesis with commit hash `91e39b86`. The Silver baseline version that was used has commit hash `fc06df18`, and was compared against the final version with commit hash `c7ae6e83`. The tests were run on a 4-core Intel Xeon E3-1240, running Ubuntu 20.04 LTS with 24GB of system memory. We used SBT version 1.4.4, Java version 11.0.7, and Z3 version 4.8.7. In the final version, the option to recover from all (pure) assertions was enabled. We tested in total 911 files: 653 from the Silicon test suite and 258 tests generated by different Viper frontends. The frontend tests were included because they contain some larger programs. Silicon includes a test runner that repeatedly verifies a set of files, trims the slowest and fastest run for each file, and averages the other run's verification times. We ran the test runner with 5 repetitions per file.

The mean relative increase of the verification time was 2.6%, and the median relative increase -0.1%. 525 files verified faster than the baseline, 379 files verified slower. Figure 6.1 shows the relative and absolute verification time differences. Although on average, verification times increased, more than half of the files actually verified faster. In theory, none of the changes introduced should have lead to a faster verification. This leads to the conclusion that the verification of most files is not affected by the changes, and that the small differences in verification times are most likely due to other reasons (e.g. CPU scheduling).

There were six files with a relative increase of verification times greater

than 50%, as shown in Figure 6.3. Four of those files also had an increased relative error output of at least 50%. Because the new version of Silicon verifies every branch and recovers from pure assertions, a larger portion of the programs were actually verified. Therefore, an increase in verification times was expected an unavoidable. Especially for programs with a lot of branching, the changes introduced can lead to higher verification times.

[1]Some of those errors occurred erroneously, due to another issue (154) in Silicon.

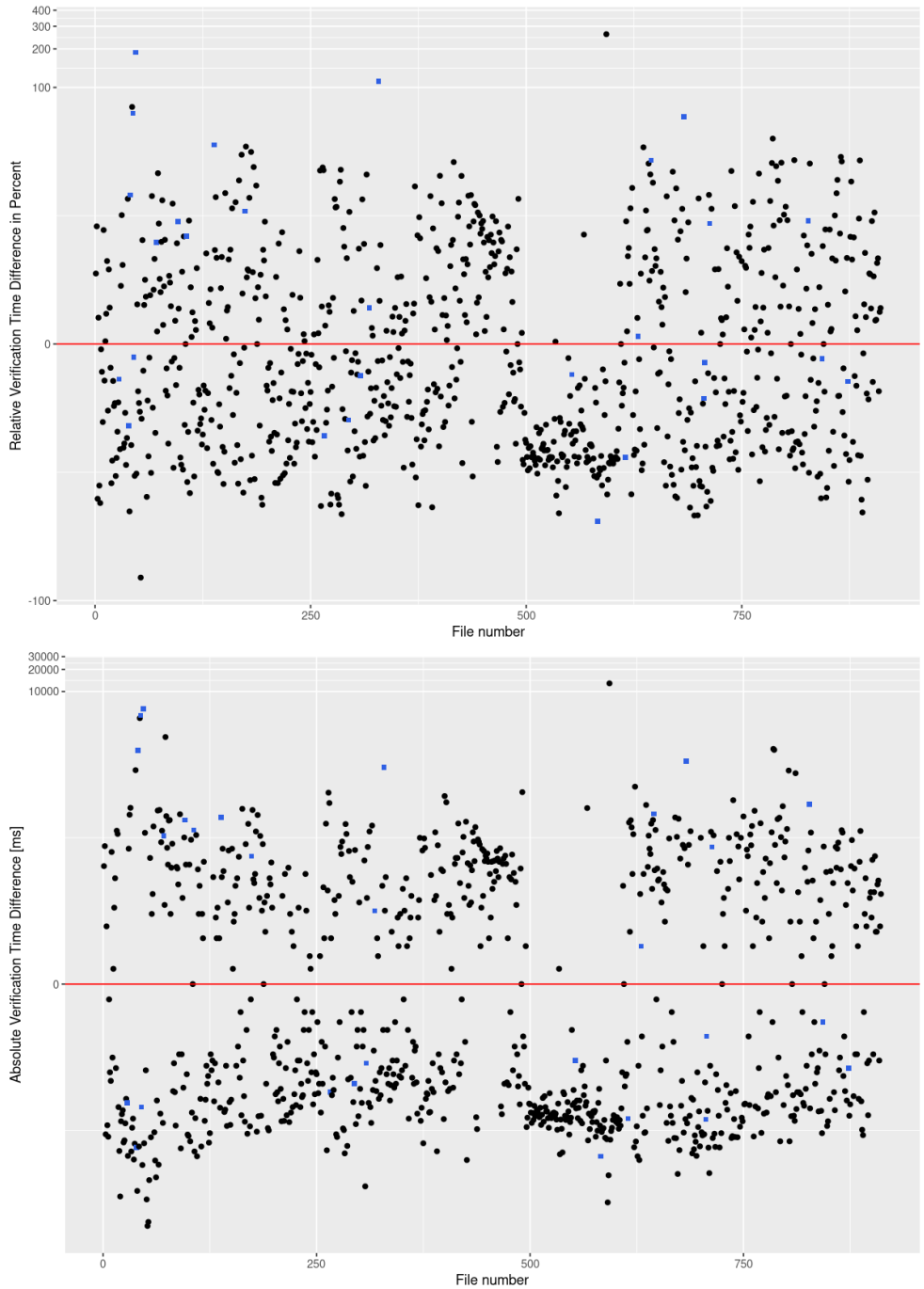

Figure 6.1: Relative and absolute verification time difference per file between the old and new Silicon version. Blue squares represent a file that had an increased error output with the new version. Dots below the red line represent files that verified faster with the new version, and dots above files that verified slower. The y-axes are scaled logarithmically.

	Old Silicon	New Silicon
Average runtime [ms]	1385	1411
Standard deviation [ms]	3070	3101
Runtime of all tests [s]	1262	1286
Number of errors reported	993	1050

Figure 6.2: Runtime and number of errors reported comparison between the old and new version of Silicon.

File	Verification Time Old Silicon [ms]	Verification Time New Silicon [ms]	Number of Errors Reported Old Silicon	Number of Errors Reported New Silicon
testssifverificationtest_lowval.py.vpr	6137	10462	7	7
testssifverificationtest_threads.py.vpr	7466	12172	4	6
348_cgmath_point.rs.vpr	3094	8917	2	7
issues/silicon/0154-1.vpr	823	1742	9	22[1]
third_party/stefan_recent/testTreeWandE2.vpr	4969	17989	1	1
sequences/bsearch.vpr	1891	3012	1	2

Figure 6.3: Runtime and number of errors reported comparison for the six files with a relative runtime increase of more than 50%.

Chapter 7

Conclusion

The goal of this thesis was to enable Silicon to recover from verification failures. This was achieved partly by enabling Silicon to explore every path resulting from branching over conditionals, and partly by enabling Silicon to recover from failed pure assertion. While changing Silicon's branching behavior such that every path is explored isn't *recovering* strictly speaking, it was a relatively simple but effective way to increase the portion of the program that is actually verified. The changes implemented, for the most part, didn't have a big impact on performance. The (albeit small) increases in verification time were expected, simply because a larger portion of the program is actually verified. A possible strategy to recover from failed impure assertions was formally presented, but not implemented.

Additionally, some usability improvements were made: Silicon can now report under which branch conditions an error occurred. It is also possible to set the number of errors after which Silicon will stop the verification.

Chapter 8

Future Work

This thesis presents the following opportunity for follow-up work: Implementing the changes proposed in Chapter 5 would enable Silicon to recover from failed impure assertions. The proposed solution however does not support Predicates, Quantified Permissions, or Magic Wands, and would need to be adapted to do so.

Bibliography

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 115–137, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[2] Tobias Brodmann. Advancing non-standard permission utilisation in program verification. Bachelor's thesis, ETH Zurich, 2018.

[3] Programming Methodology Group. Viper tutorial. https://viper.ethz.ch/tutorial/. Accessed: 2021-08-10.

[4] Programming Methodology Group. Viper website. https://viper.ethz.ch/. Accessed: 2021-08-10.

[5] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Comput. Surv.*, 8(3):331–353, September 1976.

[6] Stefan Heule. Verification condition generation for the intermediate verification language sil. Master's thesis, ETH Zurich, 2013.

[7] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

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

[9] Robin Sierra. Towards customizability of a symbolic-execution-based program verifier. Bachelor's thesis, ETH Zurich, 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

RECOVERING FROM VERIFICATION FAILURES

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
STUCK	JOSUA

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date	**Signature(s)**
Zürich 05.09.21	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.