ETH zürich

Theory-Based Verification Condition Encoding

Master Thesis

Johannes Gasser

December 8, 2025

Advisors: Prof. Dr. P. Müller, Dr. M. Eilers

Department of Computer Science, ETH Zürich

Abstract

SMT-based program verifiers like Silicon, the symbolic execution backend of the Viper verification infrastructure, encode features using uninterpreted functions with quantified axioms, relying on e-matching for instantiation. This approach suffers from well-known completeness and performance limitations due to the undecidability of quantified formulas.

This thesis investigates whether modern SMT solvers' native support for interpreted functions, algebraic data types, and built-in theories enables a viable alternative encoding that reduces or eliminates dependence on quantifier instantiation heuristics in Silicon. We develop and implement theory-based encodings for many Viper language features, including a novel approach using recursive boolean definitional functions for Viper functions and native SMT theories for built-in data types. Our evaluation demonstrates that this approach is feasible. Although e-matching remains the most reliable instantiation strategy overall, the theory-based encoding produces significantly more sound verification error results (sat instead of unknown responses), exhibits significantly increased robustness without quantifier instantiation heuristics, and achieves competitive performance for many programs.

Contents

Contents ii

1 Introduction 1

2 Background 4
- 2.1 Viper . . . 4
  - 2.1.1 Language Features . . . 4
  - 2.1.2 Quantifiers . . . 8
  - 2.1.3 Separating Conjunction . . . 10
- 2.2 Silicon: A Symbolic Execution Verifier . . . 10
- 2.3 SMT Solvers . . . 11
  - 2.3.1 SMT-LIB . . . 12
  - 2.3.2 Uninterpreted Functions . . . 12
  - 2.3.3 Asserts . . . 12
  - 2.3.4 Quantifiers and E-Matching . . . 13
  - 2.3.5 Incremental Solving . . . 14

3 Quantifiers in Silicon 15
- 3.1 Functions . . . 16
  - 3.1.1 Phase One . . . 17
  - 3.1.2 Phase Two . . . 18
  - 3.1.3 Example Function Encoding . . . 19
  - 3.1.4 Function Verification Order . . . 20
- 3.2 Sort Wrappers . . . 20
- 3.3 Domains . . . 21
  - 3.3.1 Algebraic Data Types . . . 22
  - 3.3.2 Built-in Data Types . . . 23
  - 3.3.3 Termination Plugin . . . 24
  - 3.3.4 Impure Assume Statements . . . 25
- 3.4 Magic Wands . . . 26

3.4.1 Magic Wand Snapshot Functions 26
3.4.2 Applying Magic Wands 27
3.5 User-Provided Quantifiers 27
3.5.1 Translation to SMT-LIB 27

4 Alternative SMT-LIB representations 29
4.1 SMT Solver Choice 29
4.2 Functions 30
4.2.1 Encoding Specifications Without Quantifiers 30
4.2.2 Function Verification 35
4.2.3 Complete Example 36
4.3 Sort Wrappers 37
4.4 Domains 39
4.4.1 Algebraic data types 39
4.4.2 Sequences 39
4.4.3 Sets 41
4.4.4 Termination Plugin 41
4.4.5 Impure Assume Statements 42
4.5 Magic Wands 42
4.6 User-Provided Quantifiers 43
4.7 Quantifier Landscape 44

5 Evaluation 46
5.1 Quantifier Heuristics 47
5.1.1 Candidate Configurations 47
5.1.2 Results 48
5.1.3 Less Complete Results & Timeouts 50
5.2 Encoding 51
5.2.1 Non-Recursive Functions 51
5.2.2 Recursive Functions 51
5.2.3 Sequences 52
5.2.4 Sets 54
5.2.5 Algebraic Data Types 54
5.3 Runtime 54
5.4 SMT-Solver Answers 56

6 Conclusion 59
6.1 Summary of Contributions 59
6.2 Feasibility of a theory-based approach 61
6.3 Future Work 61
6.3.1 Improved Recursive Function Handling 61
6.3.2 Alternative Sequence Encoding 62
6.3.3 Support for Additional Features 62
6.3.4 Solver Integration and Configuration 63

6.3.5 Further Tests with Viper Frontends 63
6.4 Final Remarks 64

Bibliography **65**

Chapter 1

Introduction

Deductive program verification has become a powerful tool in software engineering, enabling formal proofs of correctness for safety-critical systems. Viper, a verification infrastructure for permission-based reasoning, relies on its symbolic execution backend, Silicon, to encode verification conditions into SMT-LIB (Satisfiability Modulo Theories) formulas that are then checked by SMT solvers such as Z3 or CVC5. Like most SMT-based verifiers, Silicon's encoding strategy heavily depends on **uninterpreted functions** and **quantified axioms** to represent program semantics. These quantifiers are instantiated during SMT solving through a technique called **e-matching**, which selectively instantiates quantified formulas based on pattern matching against terms in the solver's context.

Although e-matching has proven effective for many verification tasks, it suffers from well-known limitations that can impact both completeness and termination of verification. These challenges stem from the fact that the satisfiability checking of quantified formulas is undecidable, in general, forcing SMT solvers to rely on heuristic instantiation strategies.

Modern SMT solvers like CVC5 and Z3 offer native support for algebraic data types (ADTs), **built-in theories**, such as the theories of sequences and sets, with operations like concatenation and membership, and **interpreted recursive function** definitions. These features enable expressing program semantics more directly, potentially avoiding the trigger-based instantiation challenges inherent in axiomatized encodings.

These limitations raise a fundamental question that we aim to answer with this project:

Is it feasible to reduce or eliminate dependence on e-matching by using an SMT-LIB encoding based on interpreted functions and native SMT theories instead of uninterpreted functions and quantified axioms?

Contributions

This thesis explores the feasibility of a theory-based SMT-LIB encoding for Silicon by systematically replacing quantified axiomatizations with quantifier-free or quantifier-reduced alternatives wherever possible. Our main contributions are as follows.

(1) **Analysis of quantifier usage in Silicon** (Chapter 3): We document where and how quantifiers are introduced in Silicon's current encoding, examining functions, sort wrappers, domains (including ADTs, sequences, sets, and the termination plugin), magic wands, and user-provided quantifiers. This analysis reveals that Viper's language features are predominantly encoded as uninterpreted functions with universally quantified axioms, creating a verification landscape that critically depends on trigger-based e-matching.

(2) **Alternative theory-based encoding** (Chapter 4): We present an alternative encoding that eliminates or reduces the quantifier dependence for many features of Silicon.

- Encoding Viper functions as uninterpreted functions paired with recursive boolean definitional functions that are manually instantiated at the call site.
- Replacing sort wrapper axioms with algebraic data type constructors and destructors whose properties are built into the SMT-LIB semantics
- Utilizing native theories for sequences and sets instead of domain-based axiomatizations
- Implementing a quantifier-free magic wand lookup encoding through chained function definitions
- Translating ADTs to SMT-LIB's native data types

(3) **Evaluation** (Chapter 5): We evaluate our encoding against the original Silicon on a test suite, exploring different quantifier instantiation heuristics beyond e-matching.

- Although e-matching is still the most reliable quantifier instantiation heuristic in terms of correct verifications, the encoding is more resilient than the original Silicon in the absence of quantifier heuristics
- The encoding produces significantly more `sat` (decidable) responses instead of `unknown` responses, enabling greater confidence in reported verification errors
- Performance is competitive for most cases, though recursive functions and sequence operations require further optimization

Our work demonstrates that a theory-based encoding that leverages interpreted functions and native SMT theories is not only feasible but offers meaningful

advantages in completeness and decidability for substantial subsets of Viper programs. Although challenges remain, particularly for recursive functions and certain sequence operations, the results suggest that modern SMT solvers' built-in theories provide a viable alternative to traditional axiomatization approaches.

Chapter Overview

The remainder of this thesis is organized as follows.

Chapter 2 (Background) introduces the foundational concepts necessary to understand our work. We present the Viper verification infrastructure and its Silicon backend, explain SMT solvers and the SMT-LIB language, and discuss quantifier instantiation techniques with a focus on e-matching and its limitations.

Chapter 3 (Quantifiers in Silicon) provides a systematic analysis of quantifier usage in Silicon's current encoding. For language features such as functions, sort wrappers, domains, magic wands, and user-provided quantifiers, we document the uninterpreted functions and quantified axioms generated during SMT-LIB encoding. This analysis establishes the baseline against which our alternative encoding is developed and evaluated.

Chapter 4 (Alternative SMT-LIB Representations) presents our theory-based encoding strategy. Following the structure of Chapter 3, we describe alternative implementations for each feature that eliminate or reduce the quantifier dependence. We discuss the choice of CVC5 as our target solver, detail the encoding of functions using recursive definitions, present the data type-based sort wrapper encoding, and describe the utilization of built-in theories for sequences and sets.

Chapter 5 (Evaluation) evaluates the effectiveness of the new encoding compared to the original Silicon. We examine the performance of different quantifier instantiation heuristics, analyze completeness and runtime characteristics, identify strengths and weaknesses of specific encodings, and discuss the distribution of `sat`, `unsat`, and `unknown` solver responses.

Chapter 2

Background

This chapter introduces the foundational concepts and technologies necessary to understand the SMT-LIB encodings of Silicon, a backend of the Viper verification infrastructure, presented in Chapters 3 and 4. We begin with an overview of program verification by introducing the Viper verification infrastructure and the Silicon backend. We then discuss SMT solvers and their role in verification with a focus on quantifier instantiation using e-matching.

2.1 Viper

Program verification is the process of formally proving that a program satisfies its specification. Unlike testing, which checks program behavior on specific inputs, verification aims to prove correctness for all possible inputs. This is particularly valuable for safety-critical systems where bugs can have severe consequences.

Viper is a verification infrastructure consisting of the following parts:

(1) An intermediate language that allows one to encode formal specification as part of a program.

(2) Backends that verify the specifications encoded in the intermediate verification language.

(3) Frontends for specific programming languages, including Prusti for Rust [1], Nagini for Python [2], and Gobra for Go [3].

It provides a foundation for building program verifiers for various programming languages through its flexible permission-based reasoning.

2.1.1 Language Features

Viper programs consist of methods, functions, predicates, fields, and domains.

```
1 field value: Int
2 field next: Ref
```

Figure 2.1: Globally declared fields, accessible through all refs

```
1  field value: Int
2  method swap(a: Ref, b: Ref)
3    requires acc(a.value) && acc(b.value)
4    ensures acc(a.value) && acc(b.value)
5    ensures a.value == old(b.value) && b.value == old(a.value)
6  {
7    var tmp: Int := a.value
8    a.value := b.value
9    b.value := tmp
10 }
```

Figure 2.2: A swap method exchanging the values at two heap locations

Fields, References and Permissions

Viper uses the `Ref` type to define heap-allocated objects with globally declared fields. Fields, as shown in Figure 2.1 can be accessed through all refs. For example, if we have an object variable `x: Ref`, we can access the fields `x.value` and `x.next`.

The field `x.f` is treated as a resource by Viper. To access or modify the value of `x.f`, the program state needs to hold permissions to the field. The expression `acc(x.f)` encodes the permission to access the field `f` of object `x`. Viper also supports fractional permissions to distinguish between read and write access. The maximum amount of permissions on a field is one, which allows full read-write access. Any permission amount greater than zero gives its holder read access.

```
1 requires acc(x.f, 1/2)   // Read permission (half)
2 requires acc(x.f, 1)     // Write permission (full)
```

Methods

Methods are imperative code blocks annotated with pre- and postconditions. The `requires` keyword specifies a precondition that has to hold before the method executes. The `ensures` keyword specifies a postcondition that has to hold after the method executes. In Figure 2.2 we show an example method reasoning about heap objects.

Permissions encoded in a method's preconditions are *inhaled*, i.e., added to the program state before method execution, whereas permissions in the postconditions are *exhaled*, i.e., given up from the program state after the method finishes executing. On the caller side, the caller has to exhale the required permissions when calling the method and inhales all permissions the method exhales.

```
1  function abs(x: Int): Int
2    ensures result >= 0
3    ensures result == x || result == -x
4  {
5    x >= 0 ? x : -x
6  }
```

Figure 2.3: A simple function calculating the absolute value of an integer x

Functions

Functions are parameterized, potentially recursive expressions, annotated with pre- and postconditions. We present an example function in Figure 2.3. Note the absence of a return statement, since a function encodes only a single expression.

Unlike methods, functions can appear in assertions and other specifications. They cannot modify the heap, but they may reason about resources. Reasoning about resources in functions requires specifying their permissions in the preconditions. Since functions define Viper expressions, they can be considered as side-effect-free. However, since heap references can be included in these expressions, they are generally not pure.

Predicates

Predicates are packed and named permission assertions. Their definitions can be recursive, allowing them to encode permissions and properties of recursive heap structures such as linked lists and trees. Predicates are treated as a kind of resource in Viper programs, meaning we can reason about access to predicates similar to field accesses.

```
1  predicate list(x: Ref) {
2    acc(x.value) && acc(x.next) &&
3    (x.next != null ==> list(x.next))
4  }
```

Figure 2.4: Example of a recursive predicate reasoning about a list data structure

Their encoded permission assertions are not directly present in the program state; instead, one holds the predicate as a resource. The `unfold` operation exchanges the predicate resource for the resources encoded in its body and asserts the packed assertion, while the `fold` operation does the reverse. To `unfold`, one must hold the permissions to the predicate, while the `fold` operation requires all encoded permissions to be present in the program state.

Magic Wands

Magic wands are an operator that enables reasoning about partial data structures. A magic wand assertion $A \mathrel{-\!\!*} B$ represents a resource that, together with assertion A, can be exchanged for assertion B. Both A (the left-hand side) and B (the right-hand side) can include permission assertions and logical assertions.

The two fundamental operations on magic wands are:

(1) **Package Operation:** The `package` statement creates a wand instance $A \mathrel{-\!\!*} B$ by removing resources from the current program state. Specifically, packaging requires the verifier to prove that the resources currently held, combined with the left-hand side A, are sufficient to establish the right-hand side B. The resources that remain after removing A from the current state are permanently given up and stored within the wand instance. After packaging, the program state loses these resources but gains the wand itself as a resource.

(2) **Apply Operation:** The `apply` statement consumes a wand instance $A \mathrel{-\!\!*} B$ and the left-hand side assertion A from the current state, and produces the right-hand side assertion B. Both the wand resource and the resources specified in A are removed from the program state, and the resources specified in B are added. This operation allows recovering the resources that were given up during packaging, along with any additional guarantees encoded in B.

Magic wands are particularly useful for reasoning about unbounded recursive data structures during iterative traversals. Figure 2.5 demonstrates using magic wands to traverse a linked list and compute the sum of all values while maintaining ownership of the entire structure.

In this example, the magic wand encodes the permissions to the list prefix visited so far. The wand's left-hand side represents the remaining list starting at the current node, while the right-hand side represents the complete list from the head. During each iteration, the method processes the current value, packages a new wand by folding the visited node and applying the old wand, then advances to the next node. After the loop completes and `xs` is null, applying the wand restores full ownership of the original list.

Domains

Domains define potentially polymorphic custom types and axiomatized functions. An example domain is shown in Figure 2.6. It axiomatizes an array type with three functions.

Domains provide a mechanism for encoding complex data structures and theories using uninterpreted functions and globally asserted axioms. Functions in domains are uninterpreted, meaning they have no body and are free of specifications.

```
field val: Int
field next: Ref

predicate List(xs: Ref) {
  acc(xs.val) && acc(xs.next) && (xs.next != null ==> List(xs.next))
}

method sum_list(ys: Ref) returns (sum: Int)
  requires ys != null && List(ys)
  ensures List(ys)
{
  var xs: Ref := ys
  sum := 0

  package (xs != null ==> List(xs)) --* List(ys)

  while (xs != null)
    invariant xs != null ==> List(xs)
    invariant (xs != null ==> List(xs)) --* List(ys)
  {
    var zs: Ref := xs
    unfold List(xs)
    sum := sum + xs.val
    xs := xs.next

    package (xs != null ==> List(xs)) --* List(ys) {
      fold List(zs)
      apply (zs != null ==> List(zs)) --* List(ys)
    }
  }

  apply (xs != null ==> List(xs)) --* List(ys)
}
```

Figure 2.5: Using magic wands to traverse an unbounded linked list

```
domain Array[T] {
  function array_get(a: Array[T], i: Int): T
  function array_set(a: Array[T], i: Int, v: T): Array[T]
  function array_len(a: Array[T]): Int

  axiom get_set {
    forall a: Array[T], i: Int, v: T ::
      {array_get(array_set(a, i, v), i)}
      array_get(array_set(a, i, v), i) == v
  }
}
```

Figure 2.6: Excerpt of a domain axiomatization for a polymorphic array type

2.1.2 Quantifiers

Viper supports both universal (`forall`) and existential (`exists`) quantifiers that allow expressing properties over unbounded domains. Quantifiers can appear in multiple contexts within Viper programs:.

- Domain axioms for defining properties of custom data structures
- Method preconditions and postconditions, for example to specify contracts over unbounded data structures

- Assertions, assumptions and loop invariants within method bodies to reason about loop invariants and data structure properties

A quantified expression consists of one or more bound variables with their types, an optional trigger specification, and a body expression.

Figure 2.7 demonstrates a binary search algorithm that uses quantifiers to specify the order of the sequence in the precondition and states the absence of the search key, if no result is found (`index == -1`). The method searches for a key in a sorted sequence and returns its index if found, or `-1` otherwise.

```
1   method binary_search(s: Seq[Int], key: Int) returns (index: Int)
2     requires forall i: Int, j: Int ::
3       0 <= i && j < |s| && i < j ==> s[i] < s[j]
4     ensures -1 <= index && index < |s|
5     ensures 0 <= index ==> s[index] == key
6     ensures index == -1 ==>
7       (forall i: Int :: 0 <= i && i < |s| ==> s[i] != key)
8   {
9     var low: Int := 0
10    var high: Int := |s|
11    var mid: Int

13    index := -1
14    while (low < high)
15      invariant 0 <= low && low <= high && high <= |s|
16      invariant forall i: Int ::
17        (0 <= i && i < |s| && !(low <= i && i < high)) ==> s[i] != key
18    {
19      mid := (low + high) \ 2
20      if (s[mid] < key) {
21        low := mid + 1
22      } else {
23        if (key < s[mid]) {
24          high := mid
25        } else {
26          index := mid
27        }
28      }
29    }
30    index := -1
31  }
```

Figure 2.7: Binary search using quantifiers to specify sortedness and search properties

The quantifiers in this example serve multiple purposes. The precondition quantifier (lines 2-3) expresses that the sequence is sorted for all pairs of indices. The postcondition quantifier (lines 6-7) ensures that when the key is not found, it does not exist anywhere in the sequence. The loop invariant quantifier (lines 16-17) maintains that all elements outside the current search range are not equal to the key, enabling the verifier to prove the postcondition when the loop terminates.

Triggers

Triggers guide the SMT solver's quantifier instantiation through e-matching (see Section 2.3.4). In Viper, triggers are specified explicitly using curly braces:

```
1 axiom get_set {
2   forall a: Array[T], i: Int, v: T ::
3     {array\_get(array_set(a, i, v), i)}
4     array_get(array_set(a, i, v), i) == v
5 }
```

The trigger {`array_get(array_set(a, i, v), i)`} indicates that this axiom should be instantiated whenever a similar expression is found in the current context. When triggers are not explicitly provided, Silicon automatically infers them based on the quantifier body, though automatic selection may not always choose optimal patterns. Multiple triggers can be specified for a single quantifier, and the quantifier will be instantiated if any of the patterns match.

2.1.3 Separating Conjunction

In Viper, the conjunction operator `&&` acts as a separating conjunction when combining permission assertions. That means that when multiple permissions to the same field appear in a conjunction, their permission amounts are summed up. For example, the expression `acc(x.f, 1/2) && acc(x.f, 1/2)` is equivalent to `acc(x.f, 1)`, where the two half-permissions combine to form a full permission. This additive behavior allows specifications to precisely track fractional permissions across method boundaries.

This permission model enables modular reasoning and prevents aliasing-related errors.

2.2 Silicon: A Symbolic Execution Verifier

Silicon is Viper's symbolic execution backend that verifies programs by exploring execution paths and encoding verification conditions in SMT-LIB [4].

Symbolic Execution

Symbolic execution executes programs with symbolic values rather than concrete inputs. Variables hold symbolic expressions representing sets of possible values.

When the program reaches a branch:

```
if (x > 0) {
  // then branch
} else {
  // else branch
}
```

Silicon explores both paths separately, adding path conditions:

- Then branch: assume `x > 0`

- Else branch: assume `x <= 0`

For each path, Silicon verifies that all assertions hold and specifications are satisfied.

Heap Encoding

Silicon encodes the heap using symbolic *snapshots*. A heap snapshot captures the state of memory at a particular point in the program.

Silicon uses a `Snap` sort in SMT-LIB to represent heap snapshots. This sort is structured as an algebraic data type with constructors:

- `Snap.unit`: Represents an empty heap snapshot
- `Snap.combine(l: Snap, r: Snap)`: Combines two snapshots into a larger snapshot

To embed values of different types (integers, booleans, references, etc.) into the `Snap` sort, Silicon uses *sort wrappers*. For each type `T`, Silicon generates the following sort wrapper functions.

- `TToSnap(x: T): Snap` - wraps a value of type `T` into a snapshot
- `SnapToT(s: Snap): T` - extracts a value of type `T` from a snapshot

These wrappers enable Silicon to uniformly represent all heap values while maintaining type information through the encoding and decoding functions.

2.3 SMT Solvers

SMT solvers are automated decision procedures that determine the satisfiability of logical formulas with respect to combinations of background theories. Unlike pure SAT solvers that work only with Boolean logic, SMT solvers can reason about theories such as integers, real numbers, arrays, bit-vectors, and other domains relevant to program verification.

The two most widely used SMT solvers in program verification are:

- **Z3** [5]: Developed by Microsoft Research, known for robust quantifier handling
- **CVC5** [6]: The successor to CVC4, offers strong support for various theories including strings, sequences, and sets

Both solvers support the SMT-LIB standard and provide extensive coverage of the theory, although they differ in performance characteristics and implementation of the theory.

2.3.1 SMT-LIB

SMT-LIB is the standard input language for SMT solvers [7]. It provides a common format for expressing logical formulas and allows different solvers to be used interchangeably.

The key components of SMT-LIB scripts include:

- **Sort declarations**: Define custom types
- **Function declarations**: Declare functions and constants
- **Assertions**: Logical formulas that constrain the solution space
- **Check-sat commands**: Ask the solver to determine satisfiability

2.3.2 Uninterpreted Functions

Uninterpreted functions are parameterized symbols that have no predefined meaning, except for their argument and return types. The solver only assumes that uninterpreted functions are well-defined, meaning equal inputs produce equal outputs. They are useful for modeling functions whose implementation is unknown or too complex to encode directly. Their semantics can be constrained using assertions.

2.3.3 Asserts

Asserts are formulas asserted to be true on the SMT-Level. They establish general properties that must be maintained in our SMT context, including the semantics of uninterpreted functions, or establish general properties. They extend the solver's knowledge beyond built-in theories.

For example, consider the assertions in Figure 2.8 which define that function `f` computes the absolute value.

```
1 (declare-fun f (Int) Int)
2 (assert (forall ((x Int)) (>= (f x) 0)))          ; f(x) always
      non-negative
3 (assert (forall ((x Int)) (or (= (f x) x)         ; f(x) equals x or -x
4    (= (f x) (- x)))))
```

Figure 2.8: Assertions axiomatizing the behaviour of function f

The `(check-sat)` command instructs the SMT solver to search for an assignment to all variables in the context that satisfy all assertions. If the solver finds such an assignment, it reports `sat`. If no such assignment exists, it reports `unsat`. Since the SMT problem is undecidable, there is a third possible answer, namely `unknown`, which states that the SMT solver found a potential model but is unsure if it satisfies all constraints since we are in an undecidable logic.

Quantifiers are a part of SMT-LIB that is undecidable. A solver will answer `unknown` if it encounters non-trivial quantified formulas and no complete instantiation heuristic is present.

2.3.4 Quantifiers and E-Matching

Quantified formulas allow one to express properties over unbounded domains. Universal quantifiers ($\forall$) state that a property holds for all possible values of a sort, while existential quantifiers ($\exists$) assert that there exists at least one value in the sort satisfying a property.

The satisfiability of quantified formulas is, in general, undecidable. Intuitively, SMT solvers cannot simply enumerate all possible values for quantified variables, as this would require infinite instantiations for infinite domains like integers.

Instead, solvers use heuristic instantiation strategies to selectively instantiate quantifiers with concrete terms that appear relevant to the current proof search.

E-Matching

Equality Matching (E-matching) is a quantifier instantiation technique used by SMT solvers [8]. It is the instantiation technique Silicon relies on for instantiating quantifiers during SMT solving. The technique selectively instantiates quantified formulas based on pattern matching against terms in the solver's current context.

Triggers

A trigger (or pattern) is an explicitly given term that specifies when a quantified formula should be instantiated by the e-matching algorithm. When the SMT solver encounters a term that matches the trigger pattern, it instantiates the quantifier with the matching values.

Consider the following axiom with an explicit trigger:

```
1  (assert (forall ((x Int)) (! (>= (abs x) 0) :pattern ((abs x)))))
```

The trigger `:pattern ((abs x))` indicates that this axiom should be instantiated whenever a term of the form `(abs t)` appears in the solver's context for some term `t`.

If the solver encounters `(abs 5)`, it will instantiate the axiom with `x = 5`, yielding `(>= (abs 5) 0)`. Good trigger selection is crucial for effective verification. Triggers should include all quantified variables to ensure complete instantiation. They should not be too general, causing excessive instantiations, which would lead to timeouts. They should also not be too restrictive, causing the SMT solver to miss crucial instantiations required for proving a `unsat` result.

When triggers are not explicitly specified, solvers automatically infer them from the quantifier body. Automatic trigger selection may not always choose optimal patterns, leading to incomplete or inefficient verification.

2.3.5 Incremental Solving

Silicon uses SMT-LIB's incremental solving capabilities with push/pop operations. The push command starts a context tracking all assertions and definitions that are added after the push command. The pop command clears all assertions since the last push.

```
1 (push)            ; Save current context
2 (assert ...)     ; Add temporary assumption
3 (check-sat)       ; Check satisfiability
4 (pop)             ; Restore previous context
```

Figure 2.9: Example of the push and pop commands

This allows efficiently verifying multiple program paths by sharing common prefix constraints and backtracking when exploring alternative branches.

Chapter 3

Quantifiers in Silicon

Silicon, like most SMT-based verifiers, relies heavily on quantified formulas to encode verification conditions. These quantifiers are instantiated during SMT solving through e-matching. Although e-matching has proven to be effective for many verification tasks, it has well-known limitations that are strongly dependent on trigger selection. Instantiations can be missed because of too restrictive triggers, leading to incompleteness in the form of `unknown` results, or they can be too excessive, overwhelming the SMT solver with too many unused instantiations.

To develop a verifier that is based on theories provided by SMT solvers, we must first understand where and how Silicon encodes features that could be encoded as theories. Throughout this chapter, we aim to highlight a common pattern in Viper and Silicon. Viper's features are generally encoded as uninterpreted functions whose semantics are defined through universally quantified axioms. This creates a landscape on the SMT-LIB level that heavily relies on trigger-based instantiation through e-matching. Understanding the encoding of this landscape is essential for creating alternative constructs that are quantifier-free or rely on instantiation mechanisms other than e-matching in future versions of Silicon. This chapter provides an overview of Viper language features and Silicon encoding strategies that produce uninterpreted functions and quantified axioms on the SMT-LIB level. The chapter is organized into the following sections.

Functions (Section 3.1) Explores how Viper functions are encoded using multiple function variants and how their definition is established through quantified axioms introduced in two verification phases.

Sort Wrappers (Section 3.2) Explains Silicon's representation of heap snapshots using the `Snap` sort and the quantified axioms that enable wrapping and extracting values of all possible types.

Domains (Section 3.3) Covers user-defined data types, including algebraic data types with their constructor/destructor axiomatization and built-in collection types

(sequences, sets, multisets and maps). Besides data structures, domains also encode theories like the termination plugin's well-founded order.

Magic Wands (Section 3.4) Examines the encoding of magic wands, a powerful separation logic operator. We focus on how the magic wand snapshot functions use quantified axioms to preserve heap values across wand applications.

User-Provided Quantifiers (Section 3.5) Analyzes quantifiers written explicitly by users in domain axioms and program assertions, including their trigger specifications and translation to SMT-LIB.

3.1 Functions

Functions in Viper are annotated with pre- and postconditions that specify their expected behavior. In contrast to methods, functions may be used throughout specifications and their bodies consist only of a single expression. The function verification process ensures that all functions are well-defined and that their postconditions can be derived from the precondition and their definition. Verification of function specifications is performed before method verification.

At the SMT-LIB level, each function is represented as an uninterpreted function in three distinct variants, all declared in Silicon's SMT-LIB preamble.

Actual function

Represents the function that is used on the SMT-LIB level whenever the Viper function is applied to some arguments.

Limited version

Used as an abstraction of the actual function. It is used to prevent potential matching loops caused by recursive functions in quantified axioms [9]. It is suffixed with `%limited`.

Precondition version

This auxiliary function is used as a placeholder for a Viper function's precondition. It takes the same arguments as the Viper function and returns a boolean that, if true, states that a function's precondition holds. The function itself has no axioms defining its behavior, so the function's precondition is not axiomatized. Instead, before a function call occurs, the precondition of the function is asserted by Silicon. If it holds, the precondition version of the said function is assumed to be true by Silicon. It is suffixed with `%pre`.

During function verification, Silicon incrementally introduces axioms that add semantic meaning to the limited and actual functions. The verification process is

separated into two phases to control which axioms are introduced at each point in the verification.

3.1.1 Phase One

During phase one, Silicon checks the functions' pre- and postconditions for well-definedness constraints. These include, for example, index out-of-bounds checks for data structures such as sequences and checks for ill-defined operations like division by zero. At the end of the phase, the following axioms are introduced for the function.

Post Axiom

Asserts that the precondition implies the postcondition for the limited application. The `result` variable used to reason about the return value in the postcondition is replaced by the limited application of the function.

$$\forall p_1 : T_1, \dots p_n : T_n \;::\; \{f_{limited}(p_1, \dots p_n)\;\; f_{pre}(p_1, \dots p_n) \rightarrow \\ \texttt{let result = } f_{limited}(\dots) \texttt{ in Posts}(f)$$

where p_i is the ith parameter of the function f, T_i is the type of the ith parameter, and $Posts(f)$ is a conjunction of all postconditions of f.

The expression $\{f_{limited}(p_1, \dots p_n)\}$ is the trigger used to instantiate this axiom. So when the SMT solver encounters expressions of this form, this axiom is instantiated for concrete values of $p_1 \dots p_n$.

Limited Axiom

Specifies that $\texttt{limited(app)} = \texttt{function(app)}$ and is therefore used as an entry point when using the function somewhere on the SMT-LIB level.

$$\forall p_1 : T_1, \dots p_n : T_n \;::\; \{f(p_1, \dots p_n)\}\;\; f_{limited}(p_1, \dots p_n) = f(p_1, \dots p_n)$$

where p_i is the ith parameter of the function f and T_i is the type of the ith parameter. This axiom is triggered by applications of the actual function. Since it contains a call to $f_{limited}(p_1 \dots p_n)$, its instantiation will also instantiate the postcondition axiom.

Post Precondition Propagation Axiom

This axiom assumes that the precondition of all calls of nested functions in the postcondition of f holds. In other words, the axiom states that if the precondition of f holds, all the preconditions of calls to nested functions under the appropriate guards also hold. Consider two functions where one references the other in its postcondition:

```
function g(n: Int): Int
    requires n >= 0

function f(n: Int): Int
    ensures n >= 0 ==> result == g(n)
```

The axiom is written in a context-aware way, which mirrors the postcondition's structure to propagate the precondition only when it actually holds:

$$\forall n : Int \ :: \ \{f_{limited}(n)\} \ f_{pre}(n) \rightarrow (n \geq 0 \ \ \rightarrow g_{pre}(n))$$

This ensures that `g`'s precondition is assumed only in branches where its guard holds. For example, if the postcondition has a structure like `ensures x > 0 ? result == g(x - 1) : 0`, it would be translated into `x > 0 ?` $g_pre(x)$ `: true` The axiom is triggered by the limited function application, with all occurrences of `result` replaced by the limited function.

3.1.2 Phase Two

In phase two, Silicon verifies that the function body conforms to its specification and introduces additional axioms.

Definitional Axiom

This axiom states that, under the precondition, the function application is equivalent to the function's body, with all nested function calls replaced by a call to the limited version of that function. This prevents the definitional axiom from causing a matching loop with itself for recursive functions.

$$\forall p_1 : T_1, \ldots p_n : T_n \ :: \{f(p_1, \ldots p_n), \ldots\} \ f_{pre}(p_1, \ldots p_n) \rightarrow \\ f(p_1 \ldots p_n) = \texttt{let result = } f_{limited}(\ldots) \texttt{ in Def}(f)$$

where p_i is the ith parameter of the function f, T_i is the type of the ith parameter, and $Def(f)$ corresponds to the definition of the function f. It is triggered by the application of the actual function. Additionally, it is triggered whenever a predicate that appears in the function's precondition is unfolded, and the limited function is called. This allows for automatic unfolding of recursive function definitions in the presence of potentially recursive predicates.

Body Precondition Propagation Axiom

This axiom gives the same guarantees as the post-precondition propagation axiom, but for the function's definition instead of its postcondition. It is generated in the same way, ensuring that the precondition is propagated in a context-aware manner.

3.1.3 Example Function Encoding

Figure 3.1 shows a simple Viper function and its SMT-LIB encoding. We excluded Viper's snapshots from this example as they are not relevant here and are not necessary for demonstrating what quantifiers and functions are generated.

```
1 function fac(n: Int): Int
2   requires n >= 0
3   ensures result >= 0
4 { n < 2 ? n : n * fac(n - 1) }
```

Figure 3.1: Recursive Viper function with specification

Before verification, Silicon generates three uninterpreted functions on the SMT-LIB level as shown in Figure 3.2. After phase 1 of the function verification, Silicon

```
1 (declare-fun fac (Int) Int)                   ; actual
2 (declare-fun fac%limited (Int) Int)           ; limited
3 (declare-fun fac%precondition (Int) Bool)     ; precondition
```

Figure 3.2: The uninterpreted functions output before verification as part of Silicon's SMT-LIB preamble

emits the axioms shown in Figure 3.3. They are now used by the SMT solver for future verification.

```
1  ; Limited Axiom
2  (assert (forall ((s Snap) (n Int)) (!
3    (= (fac s n) (fac%limited n))
4    :pattern ((fac s n)))))
5  ; Post Axiom
6  (assert (forall ((n Int)) (!
7    (=> (fac%precondition n) (>= (fac%limited n) 0))
8    :pattern ((fac%limited n)))))

10 ; Post Precondition Axiom
11 (assert (forall ((n Int)) (! true :pattern ((fac%limited n)))))
```

Figure 3.3: The axioms output after phase 1.

Note that the post precondition axiom is empty in the case of `fac`, since no nested functions appear in the postcondition. For uninterpreted Viper functions, i.e., functions without a body, the axiomatization is finished after phase one. For interpreted functions like `fac`, phase two generates the axioms as demonstrated in Figure 3.4.

Whenever the function `fac` is called, the limited and definitional axioms are instantiated. The instantiation of the limited axiom causes an instantiation of the

```
; Definitional Axiom
(assert (forall ((s Snap) (n Int)) (!
    (=> (fac%precondition n)
        (= (fac s n) (ite (< n 2) n (* n (fac%limited s (- n 1))))))
    :pattern ((fac s n)))))
; Body Precondition Axiom
(assert (forall ((n Int)) (!
  (=> (fac%precondition n)
      (ite (< n 2) true (fac%precondition (- n 1))))
  :pattern ((fac%limited n)))))
```

Figure 3.4: Phase 2 output of Silicon's function verification

post, the post precondition and the body precondition axioms. The instantiation of the definitional axiom also causes an instantiation of the same axioms for the next iteration of `fac`'s recursive definition. However, the unrolling of the recursive definition stops here, since no further expressions cause an instantiation of the definitional axiom.

3.1.4 Function Verification Order

Silicon computes the height of each function, based on the program's function call graph, to determine its dependency relationships. If function A calls function B, then $\text{height}(A) > \text{height}(B)$ in the non-mutually recursive case or $\text{height}(A) = \text{height}(B)$ in case A and B are mutually recursive. This ordering allows Silicon to verify functions in a dependency-aware manner. In the basic case, this means that when function A calls function B, the axioms of function B will have been generated and can be used by Silicon to prove A's specifications. Silicon verifies functions individually. So Silicon will first finish phases one and two for a function and then continue with the next. In the case of mutually recursive functions, this means Silicon will fully process one function before the other. Therefore, one of the mutually recursive functions has to be verified without the other function's axioms.

3.2 Sort Wrappers

Silicon represents heap values using the `Snap` sort (introduced in Section 2.2). Sort wrappers embed and extract values of different types into this universal representation. For each type T_i in a Viper program, Silicon generates two axiomatized functions with quantified axioms that establish their inverse relationship.

ToSnap

The ToSnap functions wrap values into a heap snapshot. The function is axiomatized by the following axiom:

$$\forall s : Snap \ :: \ \{SnapToT_i(s)\} \ \ s = T_iToSnap(SnapToT_i(s))$$

```
1 (declare-fun SnapToInt (Snap) Int)
2 (declare-fun IntToSnap (Int) Snap)
3 (assert (forall ((x Int)) (! (= x (SnapToInt(IntToSnap x))) :pattern ((IntToSnap
      x)))))
4 (assert (forall ((x Snap)) (! (= x (IntToSnap(SnapToInt x))) :pattern ((SnapToInt
      x)))))
```

Figure 3.5: Sort Wrappers for the Int type

where T_i corresponds to a type used in a Viper program.

SnapTo

The SnapTo functions unwrap values from a heap snapshot. The function is axiomatized by the following axiom:

$$\forall x : T \ :: \ \{T_i ToSnap(x)\} \ \ x = SnapToT_i(T_i ToSnap(x))$$

where T_i corresponds to a type used in a Viper program.

Example

The example in Figure 3.5 demonstrates the encoding of sort wrappers defined for Viper's `Int` type.

3.3 Domains

Domains are an important tool for encoding custom data structures and providing global axioms in Viper. They have two main purposes. First, they are used to introduce new, possibly polymorphic, types to Viper. Second, they contain uninterpreted, global functions and axioms. Axioms can be used to assume arbitrary behavior about the Viper program. In practice, they are often used to provide quantified axioms, making them a source of quantifiers. Figure 3.9 shows an example domain defining a polymorphic type `Set` with axioms defining the interaction between the `singleton`, `isMember`, and `size` functions.

```
1 domain Set[A] {
2   function singleton(e: A): Set[A]
3   function isMember(s: Set[A], e: A): Bool
4   function size(l: Set[A]): Int
5   axiom forall e: A :: {singleton(e)} isMember(singleton(e), e)
6   axiom forall e: A :: {singleton(e)} size(singleton(e)) == 1
7   ...
8 }
9
```

Figure 3.6: Excerpt of an axiomatization of a polymorphic set type

On the SMT-LIB level, Silicon translates Viper domains into a custom sort, uninterpreted functions, and global assertions. For polymorphic domains, this encoding is done for every type instantiation that appears throughout a Viper program. For example, in the program shown in Figure 3.7, the List domain will be instantiated twice, once for the `Int` type and once for the `Bool` type:

```
1     function a(l: List[Int], i: Int): Bool { ... }
2     function b(l: List[Bool], i: Int): Bool { ... }
3
```

Figure 3.7: Example Viper code using the previously defined List domain with concrete type instantiations

Viper domains are used by Viper itself and Silicon to encode types and patterns that occur frequently throughout verification. In the following subsections, we will focus on some of them.

3.3.1 Algebraic Data Types

Algebraic Data Types (ADTs) are immutable, potentially recursive data types that allow modeling complex data structures such as linked lists, trees, and option types. They are a plugin feature in Viper that gets translated into regular Viper domains during the encoding phase. Silicon is therefore not directly aware of Viper ADTs but encounters them only in the form of translated Viper domains.

Viper ADTs are translated into domains with uninterpreted functions and quantified axioms that axiomatize constructor, destructor and discriminator properties.

An ADT is defined by specifying one or more constructors, each potentially taking arguments of various types including ADTs, enabling recursive definitions. Figure 3.8 shows a simple example of a polymorphic list ADT with two constructors.

```
1   adt List[T] {
2       Nil()
3       Cons(value: T, tail: List[T])
4   }

6   method test() {
7       var list: List[Int] := Cons(1, Cons(2, Nil()))
8       assert list.value == 1
9       assert list.tail.value == 2
10  }
```

Figure 3.8: Example of a polymorphic list ADT with constructor and destructor usage

The Viper domain encoding of ADTs introduces several components: constructors to instantiate ADTs, destructors to unpack constructor parameters, and a tag function uniquely identifying each constructor.

Constructors and Destructors

Each ADT constructor is translated into a domain function that returns a value of the ADT's domain type. For each constructor argument, Viper generates a destructor function that extracts that argument from an ADT value. For example, the `Cons` constructor in Figure 3.8 generates two destructors: `value` and `tail`, while the `Nil` constructor generates no destructor.

Injectivity Axioms: For each destructor function, Viper generates an injectivity axiom that ensures it correctly extracts the constructor argument:

$$\forall p_1 : T_1, \ldots, p_n : T_n \; :: \; \{C(p_1, \ldots, p_n)\} \;\; p_i = D_i(C(p_1, \ldots, p_n))$$

where D_i is the destructor for the i-th argument of the constructor C. The axiom is triggered by the application of the constructor C.

Exhaustiveness Axiom: An exhaustiveness axiom ensures that every ADT value must have been created by one of the ADT's constructors:

$$\forall t : \text{ADT} \; :: \; \{\text{tag}(t)\} \;\; t = C_1(D_1 1(t), \ldots D_1 n(t)) \vee \ldots \vee t = C_n(\ldots)$$

where C_i corresponds to the ith constructor of the ADT and $D_i j$ to the jth destructor of the ith constructor. The axiom is triggered by the tag function or any destructor application. It binds the ADT's value using each possible constructor with its corresponding destructors.

Discriminators

The tag function returns a unique integer identifier for each ADT value based on which constructor created it. For each constructor, a tag axiom defines the tag value for all instances created by that constructor:

$$\forall p_1 : T_1, \ldots, p_n : T_n \; :: \; \{C(p_1, \ldots, p_n)\} \;\; \text{tag}(C(p_1, \ldots, p_n)) = i$$

where i is the unique tag identifier for constructor C and $p_i : T_i$ is the ith argument of the constructor. For constructors without arguments like the `Nil` constructor in Figure 3.8, the axiom is not quantified.

3.3.2 Built-in Data Types

Viper offers **Sequences**, **Sets**, **Multisets**, and **Maps** as built-in polymorphic and immutable data types. In contrast to ADTs, these data types have no finalized encoding in the Viper language. Their encoding is left to the backend. Silicon encodes all of these data types using Viper Domains, defining their polymorphic types and containing all functionality as uninterpreted functions and axioms. These encodings demonstrate the power of an axiom-based encoding. Instead of fully defining functions, many are defined by specifying their interactions.

```
1  domain Seq[A] {
2    function concat(s1: Seq[A], s2: Seq[A]): Seq[A]
3    function length(l: Seq[A]): Int
4    axiom forall s1: Seq[A], s2: Seq[A] :: {concat(s1, s2)} length(concat(s1, s2))
       == length(s1) + length(s2)
5    ...
6  }
7
```

Figure 3.9: The sequence axiomatization defining the interaction between the `concat` and `length` functions

3.3.3 Termination Plugin

The termination plugin extends Viper with the ability to prove termination of recursive functions, methods and loops. Termination is proven by requiring the user to specify a `decreases` clause that provides a measure which strictly decreases with each recursive call or loop iteration. This measure must be based on a type with a well-founded domain, ensuring that no infinitely decreasing chains exist within the domain.

Decreases Clauses

A decreases clause can be attached to function specifications and loop invariants. Figure 3.10 demonstrates a simple recursive function with a decreases clause. The clause `decreases n` specifies that the parameter `n` serves as the termination measure, and Silicon verifies that each recursive call is made with a strictly smaller value.

```
1  function fac(n: Int): Int
2      requires n >= 0
3      decreases n
4      ensures result >= 0
5  {
6      n < 2 ? n : n * fac(n - 1)
7  }
```

Figure 3.10: Recursive function with decreases clause

Well-Founded Order Domain

The termination plugin provides a polymorphic domain `WellFoundedOrder[T]` that defines the structure for well-founded orders for arbitrary types. This domain declares two functions:

- `decreasing(x:T, y:T): Bool` – Returns true iff `x` is strictly less than `y` in the well-founded order.

- `bounded(x:T): Bool` – Returns true if `x` belongs to the well-founded subset of type `T`, e.g., for integers this is satisfied when $x >= 0$.

For each type that appears in a decrease clause, the plugin needs a type-specific domain with quantified axioms that define these functions for that type. For primitive types, built-in data structures, and ADTs, these axioms are provided by the plugin itself. When using custom domains, custom axioms must be provided by the user.

Figure 3.11 shows the axiomatization for the `Int` type. The `decreasing` axiom defines that one integer is smaller than another if it is numerically less. The `bounded` axiom establishes that non-negative integers form a well-founded set.

```
1  domain IntTerminationOrder{
2      axiom integer_ax_dec{
3          forall int1: Int, int2: Int :: {decreasing(int1, int2)}
4            (int1 < int2) ==> decreasing(int1, int2)
5      }
6      axiom integer_ax_bound{
7          forall int1: Int :: {bounded(int1)}
8            int1 >= 0 ==> bounded(int1)
9      }
10 }
```

Figure 3.11: Axiomatization of the well-founded order for integers

3.3.4 Impure Assume Statements

Assume statements are called impure when they contain access predicates. An assumption about an access predicate like `assume acc(a.f)` intuitively tells Silicon to ignore all execution traces for which we do not hold the permission to `a.f`. Conjunctions of access permissions in assume statements follow the same permission summation semantics as described in Section 2.1.3. For example, the assumption `assume acc(a.f, 1/2) && acc(a.f, 1/2)` will ignore all traces where we do not hold full permission to `a.f`, since the two half-permissions sum to one full permission. This encoding is useful for expressing aliasing scenarios during verification. Because these specifications involve permission reasoning, they cannot be directly asserted on the SMT-LIB level. Viper transforms these impure assume statements into variants that can be assumed on the SMT-LIB level by using an auxiliary domain containing helper functions and axioms.

For each number of conjunctions appearing in impure assume statements, Viper generates a corresponding helper function. The helper function for n conjuncts takes n boolean conditions, each representing an access predicate, and $n+1$ permission amounts as arguments, returning a summarized permission amount.

A helper function for a conjunction of size n has the signature:

```
function assume_helper_n(c_1: Bool, ..., c_n: Bool,
```

```
p_0: Perm, ..., p_n: Perm): Perm
```

The semantics is defined by a quantified axiom that states the function returns a conditional sum of permissions:

$$\forall c_1 : Bool, \ldots, c_n : Bool, p_0 : Perm, \ldots, p_n : Perm \ :: \\ \{\texttt{helper_n}(c_1, \ldots, c_n, p_0, \ldots, p_n)\} \\ \texttt{helper_n}(c_1, \ldots, c_n, p_0, \ldots, p_n) = p_0 + (c_1 \ ? \ p_1 : \texttt{none}) + \ldots + (c_n \ ? \ p_n : \texttt{none})$$

On the SMT-LIB level, the function is encoded as an uninterpreted function with a quantified definitional axiom quantifying over all possible arguments. The axiom is triggered whenever this function is used. This allows Silicon to assert concrete permission amounts by setting the booleans $c_1 \ldots c_n$ according to potential aliasing scenarios.

3.4 Magic Wands

Magic wands (introduced in Section 2.1.1) are encoded in Silicon using magic wand snapshot functions (MWSF). This section examines how Silicon uses quantified axioms to preserve heap values across wand applications.

Silicon encodes magic wand instances using a custom sort called *magic wand snapshot functions* (MWSF). Together with a lookup function, they enable axiomatization of wand snapshot retrieval.

3.4.1 Magic Wand Snapshot Functions

Silicon declares a global uninterpreted function `MWSF_apply` that serves as the lookup mechanism for magic wand snapshots:

```
(declare-fun MWSF_apply (MWSF Snap) Snap)
```

This function takes two arguments: a MWSF and a snapshot of sort `Snap`, representing the left-hand side heap, and returns the corresponding right-hand side snapshot.

When packaging a wand $A \twoheadrightarrow B$, Silicon performs the following steps:

(1) Creates a fresh constant `mwsf` of sort `$MWSF` to uniquely identify this wand instance

(2) Captures the right-hand side heap snapshot `rhs_snap` after proving B follows from A

(3) Asserts a quantified axiom that defines the lookup behavior for this specific wand:

```
1 (assert (forall ((snap Snap)) (!
2     (= (MWSF_apply mwsf snap) rhs_snap)
3     :pattern ((MWSF_apply mwsf snap)))))
```

The axiom states that applying `MWSF_apply` to this wand's identifier `mwsf` with any snapshot will return the captured `rhs_snap`.

3.4.2 Applying Magic Wands

When a wand is applied, Silicon evaluates expressions in the wand's right-hand side using the `MWSF_apply` function. For a wand chunk with snapshot function `mwsf` and a left-hand side snapshot `lhs_snap`, the expression `MWSF_apply(mwsf, lhs_snap)` retrieves the right-hand side heap state.

3.5 User-Provided Quantifiers

User-provided quantifiers are quantified expressions explicitly written by Viper users in their programs, as opposed to quantifiers generated internally by Silicon for encoding language features. As introduced in Section 2.1.2, quantifiers allow expressing properties over unbounded domains. All user-provided quantifiers are translated directly into quantified formulas on the SMT-LIB level, where they rely on e-matching for instantiation.

3.5.1 Translation to SMT-LIB

User-provided quantifiers in Viper are translated directly to SMT-LIB quantifiers with their trigger specifications preserved. For example, the loop invariant quantifier from Figure 2.7:

```
invariant forall i: Int ::
  (0 <= i && i < |s| && !(low <= i && i < high)) ==> s[i] != key
```

is translated to SMT-LIB as:

```
(assert (forall ((i Int)) (!
    (=> (and (<= 0 i) (< i (Seq_length s))
             (not (and (<= low i) (< i high))))
         (not (= (Seq_index s i) key)))
    :pattern ((Seq_index s i)))))
```

The `:pattern` annotation is automatically inferred by Silicon since no explicit trigger was specified in the Viper code. The pattern `(Seq_index s i)` ensures the quantifier instantiates when accessing specific indices of the sequence. Quantifiers

appearing in methods are asserted at specific program points and, through SMT-LIB's incremental solving with push/pop operations, are removed from the SMT context once they go out of scope.

Chapter 4

Alternative SMT-LIB representations

This chapter presents the implementation of theory-based alternatives in Silicon's encoding of Viper programs to SMT-LIB, wherever possible. Building on Chapter 3, which identified where and how quantifiers are introduced in the original encoding, we describe the concrete implementation strategies that eliminate or reduce dependence on quantifiers and e-matching.

The chapter closely follows the structure of Chapter 3, examining alternative implementations for all introduced features.

Functions (Section 4.2): Presents the function encoding, including mechanisms for encoding Viper's specifications in pure SMT-LIB functions.

Sort Wrappers (Section 4.3): Introduces a fully data-type-based encoding.

Domains (Section 4.4): discusses the role of domains in this new landscape that does not solely depend on e-matching and introduces several alternatives for domain-based feature encoding in Viper.

Magic Wands (Section 4.5): describes a quantifier-free encoding of the magic wand snapshot functions.

User-Provided Quantifiers (Section 4.6): addresses the role and our handling of user-provided quantifiers in a theory-based SMT-LIB landscape.

Quantifier Landscape (Section 4.7): discusses the remaining quantifiers in this alternative landscape. Includes a list of features that are now truly quantifier-free and discusses the handling of user-defined quantifiers.

4.1 SMT Solver Choice

Early during the development of our encoding, we decided to change the targeted SMT solver from Z3, used by the original Silicon, to CVC5. There are two main reasons for this decision:

(1) **Z3 rewriting of recursive functions**: Defining recursive functions in Z3 might lead to undefined behavior in some cases, since Z3's rewriter does not guard against non-terminating recursive functions. For us, this was expressed by the non-termination of Z3 when asserting formulas after a problematic, recursively defined function was defined. An example of this is shown in Figure 4.1 CVC5 correctly reports `unsat` for this example.

```
1  (define-funs-rec ((g ((a Int) (b Int)) Bool)) ((or
2    (= a b)
3    (ite (> a b) (g (- a 1) b) (g a (- b 1)))
4  )))
5  (assert (not (g 2 2)))
```

Figure 4.1: Example of a recursive function that gets stuck when adding an assertion.

(2) **More built-in theories**: CVC5 currently supports more built-in theories that match Viper's language constructs than Z3. For example, CVC5 offers a comprehensive set theory that Z3 currently lacks.

This choice does not mean that Z3 will not be supported in our encoding, but rather that the encodings we present are tested and optimized mainly for CVC5. Also, the `Set` encoding will simply fall back to the domain-based encoding.

4.2 Functions

The original Silicon implementation (Section 3.1) encodes function semantics through uninterpreted functions with quantified axioms. Our implementation shows an alternative to this approach with recursive function *definitions* using SMT-LIB's `define-fun-rec` command.

4.2.1 Encoding Specifications Without Quantifiers

The challenge in modeling Viper functions using interpreted SMT-LIB functions is that SMT-LIB's `define-fun-rec` construct only supports function *definitions*, not function *constraints*. While we can encode the computational behavior of a function, we cannot directly specify constraints about the function's behavior that go beyond its definition.

Consider a direct encoding of the `fac` function shown in Figure 4.2.

```
1  (define-fun-rec fac ((n Int)) Int
2      (ite (< n 2) n (* n (fac (- n 1)))))
3
```

Figure 4.2: Interpreted SMT-LIB function for the factorial Viper function

While preconditions could theoretically be encoded within the definition itself (e.g., using a conditional `(ite (>= n 0) <definition> <default-value>)`), it still only addresses half the problem.

In Viper, the postcondition `ensures result >= 0` expresses a property that should hold for all uses of `fac`. However, SMT-LIB provides no quantifier-free mechanism to assert such constraints separately from the function definition. We cannot express “whenever `fac(n)` appears in a formula, assume `fac(n) >= 0`” without using quantified axioms of the form:

```
1 (assert (forall ((n Int)) (=> (>= n 0) (>= (fac n) 0))))
```

Moreover, even though postconditions of interpreted functions should follow from their definitions, the SMT solver may not automatically derive these properties during verification. For complex recursive functions, the solver needs explicit guidance about what properties hold, especially when those properties are used in specifications of other functions or methods. Without a way to express post-conditions as separate constraints, we lose the modularity and abstraction that Viper’s function specifications provide. Most critically, abstract Viper functions, so functions that have only specifications but no definition, can’t be modeled at all using `define-fun-rec`, since there is no definition to encode.

Our alternative encoding addresses this by bundling the function’s definition, postcondition, and crucially, the definitions and postconditions of all nested function calls into a single recursive boolean function. When this function is asserted at a use site, the SMT solver can automatically unfold it to access the complete chain of definitions and specifications needed for verification.

In this encoding, Viper functions are divided into two SMT-LIB components: an uninterpreted function representing the function’s value, and a boolean definitional function that relates the uninterpreted function to its specification, body, and nested definitions:

```
1 (declare-fun fac (Snap Int) Int)
2 (define-fun-rec fac%def ((s Snap) (n Int)) Bool
3   (=> (>= n 0)
4       (and (= (fac s n) (ite (< n 2) n (* n (fac s (- n 1)))))
5            (>= (fac s n) 0)
6     )))
```

Figure 4.3: Example of the fac function encoded as an uninterpreted and definitional function

The definitional function follows the structure `precondition` $\rightarrow$ (`definition` $\wedge$ `postcondition`), encoding the computational behavior, postcondition guarantee, and preconditions for recursive calls. This ensures that the definition and post-condition of a function are only asserted if its precondition holds. Whenever the uninterpreted function appears in an SMT-LIB assertion, the definitional function

is asserted for the corresponding arguments. This approach closely resembles what e-matching would do. The definitional function serves as an ”axiom” that we instantiate by hand whenever we encounter a function call.

Context-Aware Nested Definitions

Unfortunately, the encoding shown in Figure 4.3 is still incomplete. It uses a naive approach that essentially boils down to simply adding an SMT-level assertion for the definition and postcondition whenever a function is used. However, this approach fails to provide the SMT solver with sufficient information to reason about *nested* function calls and *recursive* function definitions.

Consider what happens when the solver tries to verify a property about `fac(s, 5)`. Asserting = `fac%def(s, 5)` tells the solver that:

```
1  (= (fac s 5) (ite (< 5 2) 5 (* 5 (fac s 4))))
```

This reduces the question about `fac(s, 5)` to a question about `fac(s, 4)`. However, without `fac%def(s, 4)`, the solver has no information about what `fac(s, 4)` equals or what properties it satisfies. The solver cannot unfold the recursion further, making it impossible to derive that `fac(s, 5) >= 0` from the definition.

In the `fac` example, `fac%def` currently gives `fac(s, n)` an interpretation, but not `fac(s, n - 1)`. The SMT solver is therefore unable to reason about the complete recursive structure, leading to verification incompleteness.

To understand the challenge, consider where function calls can appear in SMT-LIB. At the top level of a verification task, seen in Figure 4.4, we can write separate assertions.

```
1  (assert (fac%def s 5))        ; Assert definition for fac(s, 5)
2  (assert (fac%def s 4))        ; Assert definition for fac(s, 4)
3  (assert (>= (fac s 5) 0))     ; Use the function
```

Figure 4.4: How recursive function definitions could be manually unrolled

Here, we are in an *assertion context*, a context where we can add multiple SMT-level `(assert ...)` statements to provide the solver with whatever information it needs.

However, when defining the body of `fac%def` itself, we are *not* in an assertion context. Inside a `define-fun-rec`, we can only write a single boolean expression, not multiple assertions, as demonstrated in Figure 4.5.

The same problem arises in other non-assertion contexts. For example, within a quantifier body like in Figure 4.6, we cannot insert separate assertions.

```
(define-fun-rec fac%def ((s Snap) (n Int)) Bool
  (=> (>= n 0)
      (and (= (fac s n) (ite (< n 2) n (* n (fac s (- n 1)))))
           (>= (fac s n) 0)
           ; PROBLEM: We need fac%def for (n-1) here, but cannot
           ; write (assert (fac%def s (- n 1))) inside this
    definition
      )))
```

Figure 4.5: While we can encode the evaluation logic for `fac`, we have no option to expres pre or postconditions

```
(assert (forall ((x Int))
  ; We can only write Boolean expressions
  ; Cannot add (assert (fac%def s x)) for nested function calls
  (=> (>= x 0) (>= (fac s x) 0))))
```

Figure 4.6: Definitions can't be asserted when calling a function is in a quantifier.

Similarly, inside a `let` expression, we can only bind values, not add assertions about nested function calls that appear in those values.

Our approach includes nested definitions as conjunctions within the definitional function. This changes our structure to:

$$\texttt{precondition} \rightarrow (\texttt{definition} \land \texttt{postcondition} \land \texttt{nested definitions})$$

However, simply conjoining all nested function definitions unconditionally would be unsound. The nested definitions must respect the control flow of the original function body to ensure that definitions are only included when the corresponding function calls are actually reachable.

```
field v: Int

function f(x: Ref): Int
  requires acc(x.v)
  ensures x != null

function g(x: Ref): Int
  requires x != null ==> acc(x.v)
{
    x != null ? f(x) : 10
}

method m(x: Ref)
  requires x == null
{
  var i: Int := g(x)
  assert false
}

```

Figure 4.7: By asserting f's definition unguarded, we are able to assert false in m

Consider the example in Figure 4.7. On line 20, we are able to prove false, when naively conjoining the definition of `f` to function `g`'s definition.

This happens because we unroll `f`'s postcondition unprotected into `g`. On the SMT-LIB level we can be visualized this problem by inlining the assertions that `g%def` and `f%def` would make, as demonstrated in Figure 4.8.

```
1  (declare-const x Ref)
2  ; Precondition of m
3  (assert (= x Ref.Null))

5  (declare-const i Int)
6  ; Inlined definition of g, conjoined with the inlined
    definition of f
7  (assert (and (ite (not (= x Ref.Null)) (f x) 10) (not (= x Ref.Null))))
8  ...
```

Figure 4.8: The defintions of `f` and `g` produce a contradiction

Here, we produced a contradiction since we assert `x == null && x != null` which can't be both true. This happens because `f`'s postcondition, which would hold in the context of function `f`, is unrolled into `g`'s definition.

Our solution is to extract nested function calls in a *context-aware* manner that respects short-circuit evaluation and conditional expressions. For each expression in the function body, we recursively identify all nested function calls and generate definitional assertions that are guarded by the appropriate conditions.

The transformation follows these rules:

- **Function applications**: For a call `f(e1, ..., en)`, first recursively transform all argument expressions, then add `f%def(e1, ..., en)`.
- **Conjunctions** (`A && B`): Transform `A`, then transform `B` guarded by `A`: `tr(A) && (A ==> tr(B))`. This respects short-circuit evaluation. If `A` is false, `B` is not evaluated.
- **Disjunctions** (`A || B`): Transform `A`, then transform `B` guarded by `!A`: `tr(A) && (!A ==> tr(B))`.
- **Implications** (`A ==> B`): Transform `A`, then transform `B` guarded by `A`: `tr(A) && (A ==> tr(B))`.
- **Conditionals** (`ite(c, t, e)`): Transform the condition `c`, then guard the transformations of the branches: `tr(c) && (c ==> tr(t)) && (!c ==> tr(e))`.
- **Let expressions**: Transform all bound expressions, then transform the body with the bindings in scope.
- **Quantifiers**: Transform the quantifier body and wrap it in a quantifier with the same variables and triggers.

```
1  (define-fun-rec fac%def ((s Snap) (n Int)) Bool
2    (=> (>= n 0)
3        (and (= (fac s n) (ite (< n 2) n (* n (fac s (- n 1)))))
4             (>= (fac s n) 0)
5             (ite (< n 2) true (fac%def s (- n 1)))
6     )))
```

Figure 4.9: The finalized definition of function `fac`, encoding pre and postconditions its definition and nested definitions

For the `fac` function, the recursive call `fac(s, n-1)` occurs only when `n >= 2` (the else branch of `ite (< n 2) ...`). Applying the transformation rules, the definitional function includes `fac%def(s, n-1)` guarded by `!(< n 2)`, which is equivalent to `n >= 2`. The final definition is shown in figure 4.9.

This ensures the SMT solver is aware of the recursive definition through `fac%def` and may propagate the specification throughout the recursion. For non-recursive functions calling other functions, the same principle applies. This logic already existed in Silicon to propagate preconditions for nested function calls. We adapted it to fit our new encoding approach.

4.2.2 Function Verification

Our goal is to mimic Silicon's function verification process. This means we do not emit function specifications before they are checked for well-definedness and correctness. In contrast to the original Silicon, this is slightly harder, as function definitions are immutable, meaning we cannot gradually append new axioms during the function verification process. To address this limitation, our encoding maintains multiple versions of each Viper function, mimicking the definition state of Viper functions in the old Silicon encoding.

Uninterpreted Version

An uninterpreted function that serves as a placeholder throughout the SMT-LIB script. It takes the same arguments as the Viper function and produces the same type of results.

Postcondition Version

Generated after phase one verification. It encodes only the pre- and postconditions of the function.

$$\texttt{f\%post}(s, p_0, \ldots, p_n) = (\texttt{pre} \rightarrow \texttt{posts} \wedge \texttt{nested definitions})$$

Definitional Version

Generated after phase two verification. It contains all definitions of the posts function and extends it by relating the uninterpreted function to its body.

$$\texttt{f\%def}(s, p_0, \ldots, p_n) = \texttt{pre} \rightarrow (\texttt{f}(s, \ldots) = \texttt{body} \land \texttt{posts} \land \texttt{nested definitions})$$

Mutual Recursion

To provide a complete verification of mutually recursive functions, we require the definition of all functions in the mutually recursive chain.

```
1  function foo(i: Int): Int
2  requires i >= 0
3  ensures result > 0
4  { i > 0 ? bar(i - 1) : 20 }

6  function bar(i: Int): Int
7  requires i >= 0
8  ensures result > 0
9  { i > 0 ? foo(i - 1) : 10 }
```

Figure 4.10: Example of mutually recursive functions

When verifying the function `foo`, its body references `bar`. Silicon's original function verification is incomplete in this example. It would first verify the specifications of `foo` and encounter a verification error since `bar` is still missing its specification at this point. To address this, we adapted the order in which functions are verified to a batched approach instead of one by one. The batches are grouped by function height. For each batch, first phase one of the function verification process is performed for all functions in that height, and then phase two. This means that for mutual recursion, postconditions can be used to verify specifications such as the one shown in Figure 4.10.

The `FunctionCallTransformer` needs to know which function is in which phase. When encountering a function call, it selects the most complete version for the called function. In Figure 4.10, this means that if `foo` reaches phase two first, it will use the postcondition version of `bar` for its verification, allowing it to successfully verify. Removing `bar`'s postcondition would result in a verification failure.

4.2.3 Complete Example

The factorial function from Chapter 3:

```
1  function fac(n: Int): Int
2    requires n >= 0
3    ensures result >= 0
4  { n < 2 ? n : n * fac(n - 1) }
```

generates these SMT-LIB definitions:

```
1  ; Initial declarations
2  (declare-fun fac (Snap Int) Int)

4  ; Phase 1: Silicon performs well-definedness checks

6  ; After Phase 1: Postcondition version
7  (define-fun-rec fac%post ((s Snap) (n Int)) Bool
8    (=> (>= n 0) (>= (fac s n) 0)))


11 ; Phase 2: Prove postconditions follow from body and preconditions
12 (declare-const n Int)
13 (declare-const s Snap)
14 (declare-const res Int)
15 (assert (>= n 0)) ; Pre
16 ; Assert nested postcondition for inductive reasoning
17 (assert (fac%post s (- n 1)))
18 (assert (= res (ite (< n 2) n (* n (fac s (- n 1)))))) ; Body
19 (assert (not (>= res 0))) ; Post, should report unsat
20 ...
21 ; After Phase 2: Definitional version
22 (define-fun-rec fac%def ((s Snap) (n Int)) Bool
23   (=> (>= n 0)
24       (and (= (fac s n) (ite (< n 2) n (* n (fac s (- n 1)))))
25            (>= (fac s n) 0)
26            (=> (not (< n 2))
27                (and (>= (- n 1) 0)
28                     (fac%def s (- n 1)))))))

30 ; Now fac%def can be used for more complete reasoning
31 ; Verification of other functions
32 ...
33 ; Method verification using the complete fac%def
34 ...
```

4.3 Sort Wrappers

The original implementation of sort wrappers (Section 3.2) relies on uninterpreted functions with quantified axioms to express injectivity and inverse properties. Our implementation replaces this with algebraic data type constructors and destructors.

```
1 (declare-data-types ((Snap 0)) (
2   ((Unit)
3    (Combine (First Snap) (Second Snap))
4    (IntToSnap (SnapToInt Int))
5    (BoolToSnap (SnapToBool Bool))
6    ...)))
```

Each type has a corresponding constructor (e.g., `IntToSnap`) and destructor (e.g., `SnapToInt`). The semantics of the SMT-LIB data type automatically provide injectivity and inverse properties, eliminating the need for quantified axioms. However, there is one semantic difference using SMT-LIB data types compared to the functional approach of Section 3.2.

Recall the axiomatization of the functions `SnapToInt` and `IntToSnap`. Whenever

SnapToInt was applied to a value s of type Snap on the SMT-LIB level, the axiom is instantiated, declaring s = IntToSnap(v), where v is an integer. SMT-LIB data types, however, do not automatically make such an assumption. Using the destructor SnapToInt on a value s of type Snap that was not constructed with the constructor IntToSnap, is under-specified, allowing arbitrary values for s. In certain scenarios, such as the one shown in Figure 4.11, we see an example affected by this under-specification.

```
1  field f: Int

3  predicate P(x: Ref) { acc(x.f) && x.f == 0 }

5  function fun(x: Ref): Int
6    requires acc(P(x))

8  method test(x: Ref)
9    requires acc(P(x))
10 {
11   var v1: Int; var v2: Int;

13   v1 := fun(x)
14   exhale acc(P(x))
15   inhale acc(P(x))
16   v2 := fun(x)

18   assert v1 == v2
19 }
20
```

Figure 4.11: Example of a Viper program for which SMT-LIB data types initially created an incompleteness

By exhaling and inhaling the predicate P(x), we first give up our permissions to P(x), and then claim them again. Even though the heap snapshot should still have the same structure and value, Silicon generates a new Snap variable after these operations that is implicitly passed to fun when using it on the second call. This Snap differs from the one passed to the initial application of fun On the SMT-LIB level, this produces a chain of assertions of the following form:

```
1 (assert (= (SnapToInt (first (first s1))) 0))
2 (assert (= v1 (fun s1 x)))
3 ...
4 (assert (= (SnapToInt (first (first s2))) 0))
5 (assert (= v2 (fun s2 x)))
6 ...
7 (assert (not (= v2 v1))) ; SAT
```

Here, s1 is our heap snapshot before the exhale, and s2 is the heap snapshot after the inhale statements. Since the structure of s1 and s2 is not enforced by our axioms, the SMT solver produces the following model in this example:

```
1 (define-const s1 Snap (combine (combine unit unit) unit))
2 (define-const s2 Snap (combine (combine (IntToSnap 0) unit) unit))
```

Note that for s1, the expected (IntToSnap 0) is replaced by the unit constructor. Therefore, the values s1 and s2 are not equal, and the function fun might produce different results for both applications, since it has no concrete definition.

We address this problem by explicitly letting Silicon assume the whole Snap structure recursively, whenever a snapshot is unpacked.

4.4 Domains

Because of the nature of Viper domains, which reflect the SMT-LIB encoding strategy of uninterpreted functions and axioms, there is no general translation we can apply to all domains. However, we will focus on translating the features encoded in domains into quantifier-free versions. More precisely, we focus on translating Viper's ADTs, sequences, sets, and maps, and the domains created for impure assumptions and Viper's termination plugin.

4.4.1 Algebraic data types

The implementation extends Silicon's SMT-LIB term representation to support native ADTs using the declare-data types construct. Although the translation was straightforward, the interpretation of the data types on the SMT-LIB level slightly differs from Silicon's ADT plugin.

Recursive data type definitions

Data types need to be well-founded in CVC5. Data types like the one shown in Figure 4.12 are rejected by the SMT solver because they are not considered well-founded, since nested recursion in the sequence type, contrary to simple recursion, is not considered well-founded. A workaround for this example is to simply define an empty constructor next to the Node constructor.

```
1  adt DAG[T] {
2      Node(value: T, successor: Seq[DAG[T]])
3  }
```

Figure 4.12: ADT considered not well-founded by CVC5 but accepted by original Silicon.

Although the data type is now well-founded, CVC5 sometimes still reports an error during verification that nested recursive data types are not allowed. To explicitly allow them, we set the option dt-nested-rec to true by default.

4.4.2 Sequences

CVC5 and Z3 provide a sequence theory containing operations for construction, concatenation, length, element access (seq.nth), and subsequence extraction

```
1  adt DAG[T] {
2      EmptyNode()
3      Node(value: T, successor: Seq[DAG[T]])
4  }
```

Figure 4.13: Working data type when `dt-nested-rec` is turned on.

```
1     (define-fun seq_extract_Seq<Int> ((s (Seq Int)) (i Int) (j Int)) (Seq Int)
2   (seq.extract s (ite (< i 0) 0 i) (ite (< j 0) 0 j )))
```

Figure 4.14: SMT-LIB Wrapper for the seq.extract function for Int sequences. It applies a lower bound of zero to arguments.

(`seq.extract`). Most Viper sequence operations map directly to those primitives.

Element Membership

Viper's `x in s` translates to `seq.contains(s, seq.unit(x))` since `seq.contains` checks for subsequence containment.

Subsequence Operations

Viper's `take(s, n)` and `drop(s, n)` are translated to `seq.extract(s, 0, n)` and `seq.extract(s, n, len(s))`. The SMT-LIB standard leaves `seq.extract` under-specified for negative indices or lengths, which means the SMT solver may answer with arbitrary sequences for negative indices, while Viper simply replaces the negative arguments with zero. Therefore, we provide a custom `seq_extract_<sort>` wrapper that guards against this underspecified behavior by changing negative indices to zero.

The wrapper, as shown in Figure 4.14, is generated for every sequence type used throughout a Viper program, since polymorphic functions are not yet supported by CVC5 and Z3.

Integer range

CVC5's and Z3's sequence theory lacks a built-in range function. We implemented it on the Viper level as a recursive Viper function shown in Figure 4.15. Since this function relies on a recursive definition, it cannot be considered quantifier-free.

```
1  function seq.range(i: Int, j: Int): Seq[Int]
2    ensures j > i ==> |result| == j - i
3    ensures j <= i ==> |result| == 0
4  { j > i ? [i] ++ seq.range(i+1, j) : Seq() }
```

Figure 4.15: seq.range function implemented as a recursive function

4.4.3 Sets

CVC5 provides an extensive theory of sets. All Viper set features can be directly translated into this theory, with one exception.

Set Disjointness

CVC5 offers no implementation for set disjointness checks. We implemented it by checking if the intersection of two sets is empty, as shown in Figure 4.16.

```
1  (= (as set.empty (Set T)) (set.inter s1 s2))
```

Figure 4.16: check for set disjointness

Sets in Z3

Z3 does not yet officially support a built-in set theory. Some features, as well as a sort alias, are built on SMT-LIB's array theory, which supports arrays with polymorphic indices and values. The sort `Set T` aliases the sort `(Array T Bool)`. Defining an array like `a: Array[Int, Bool]`, so an array indexed by integers with boolean values can be used to resemble a set of integers by specifying that every integer that is contained in the set has the value `true` in array `a`. Member checks to the set could then be encoded as `a[i] == true`. The main reason we could not encode Viper's complete `Set[T]` type using this approach is that arrays lack a cardinality function. Finding a suitable encoding for this function is very hard, especially if it should be quantifier-free.

4.4.4 Termination Plugin

The termination plugin defines well-founded orderings through `decreasing` and `bounded` operations. For simple types (integers, booleans, permissions), these can be directly translated into non-domain Viper functions, as shown in Figure 4.17 for the `Int` type. This new encoding is implemented directly in Viper's termination plugin rather than Silicon.

```
1  function decreasing(arg1: Int, arg2: Int): Bool {
2    arg1 < arg2
3  }
4  function bounded(arg1: Int): Bool {
5    arg1 >= 0
6  }
```

Figure 4.17: Example of the well-founded set definition for the Int type

Decreasing functions for polymorphic types, such as sequences or sets are not yet encoded. The main reason for this is that with the use of domains, Viper lacks features to model polymorphic, interpreted functions.

4.4.5 Impure Assume Statements

The original encoding used auxiliary domains with quantified axioms. Our implementation generates helper functions for conditional permission sums:

```
1  function __iar__assume_helper_n(c_1: Bool, ..., c_n: Bool,
2                                  p_0: Perm, ..., p_n: Perm): Perm {
3      p_0 + (c_1 ? p_1 : none) + ... + (c_n ? p_n : none)
4  }
```

The helper function defines exactly the axiom that was stated in 3.3.4. This was also implemented in Viper directly instead of Silicon, since impure assume statements are translated before they reach a backend.

Quantified permissions in impure assume require inverse functions mapping resource locations to quantified variables. While infrastructure exists to generate these axioms, we have not yet translated them into a function-based approach, as we deliberately omitted quantified permissions from this project.

4.5 Magic Wands

The previous MWSF encoding highlighted the power of a function encoding based on quantifiers and uninterpreted functions. In the original Silicon, we were able to provide an uninterpreted function `MWSF_apply` and an uninterpreted sort `MWSF` and gradually extend the function definition, whenever new magic wands were packed during program evaluation. As we have already highlighted in Section 4.2, this is a weakness of an encoding based on interpreted SMT-LIB functions.

```
1  (declare-sort MWSF 0)
2  ; Old Encoding
3  (declare-fun MWSF\_apply (MWSF Snap) Snap)
4  ; Add a new wand instance by defining a const \& refining the
       function with a new axiom
5  (declare-const MWSF_1 MWSF)
6  (assert (forall ((s Snap)) (= (MWSF\_apply MWSF_1 s) (rhs_snap))))

8  ; New Encoding

10 ; First link of our chain
11 (declare-fun MWSF\_apply (MWSF Snap) Snap)
12 ; Add a new wand instance by defining a const \& defining a new
       link in our chain
13 ; It checks for its dedicated const, or passes the lookup up the
       chain
14 (declare-const MWSF_1 MWSF)
15 (define-fun MWSF\_apply_1 ((m MWSF) (s Snap)) Snap (ite (= m MWSF_1) rhs_snap
       (MWSF_apply m s)))
```

In our alternative encoding, we want to provide the same functionality as the original Silicon did. We achieve this by replacing each quantified axiom emitted for `MWSF_apply` by defining a new version of the `MWSF_apply` function. Each of these definitions stores the right-hand side of one MWSF tag. If the provided tag does not match the function's tag, the previous function is called. This process is repeated until the correct tag is found, or the uninterpreted `MWSF_apply` function is hit. The current version of the `MWSF_apply` function has to be tracked by Silicon in its state variable. An example of such a function chain is shown in Figure 4.18 where a chain for two MWSFs is shown.

```
1   (declare-fun MWSF_apply (MWSF, Snap) Snap)

3   ; first packaged wand
4   (declare-const tag_1 MWSF)
5   (define-fun MWSF_apply_1 ((t MWSF) (s Snap)) Snap
6       (ite (= t tag_1) rhs (MWSF_apply(t, s))))

8   ; second packaged wand
9   (declare-const tag_2 MWSF)
10  (define-fun MWSF_apply_2 ((t MWSF) (s Snap)) Snap
11      (ite (= t tag_2) rhs (MWSF_apply_1(t, s))))
12
```

Figure 4.18: Example encoding of two packed magic wands

4.6 User-Provided Quantifiers

At the beginning of this project, one of our primary concerns was how to handle user-provided quantifiers within our new encoding landscape. Previous attempts at using the theory of sequences [10] have raised questions about the reliance on quantifier instantiation heuristics in the presence of built-in theories. Our main question was whether our encoding based on built-in theories and interpreted functions would negatively impact quantifier instantiation, potentially introducing new incompleteness or performance degradation.

We identified three possible approaches to address this concern:

(1) **Bounded data structures**: Attempt to automatically bound quantifiers over data structures such as sequences, sets, or maps. This would involve analyzing the quantifier's domain and replacing infinite domains with finite approximations wherever possible.

(2) **Manual instantiation heuristics**: Implement custom heuristics to detect bounded quantifiers of the form `forall i: Int :: 0 <= i && i <= 10 ==> ...` and manually unroll or instantiate them. This approach would require pattern matching on the quantifier structure to identify bounded ranges and generate explicit instantiations.

(3) **Direct encoding**: Leave user-provided quantifiers as they are in the SMT-LIB output, relying on the SMT solver's native quantifier handling capabilities.

To determine which approach would be most effective, we closely observed the behavior of Silicon's extensive test suite. The results were encouraging: the direct encoding approach (option 3) proved to be sufficient for almost all tested scenarios.

Modern SMT solvers, particularly CVC5, demonstrate robust handling of user-provided quantifiers even in the presence of interpreted functions and built-in theories. The solver's internal instantiation strategies work well with our encoding, and we observed no significant performance degradation or incompleteness issues in most cases.

The one notable exception to this positive result involves a few quantifiers defining triggers that use built-in functions of the sequence theory. However, this limitation seems to affect a relatively small subset of verification tasks and does not significantly impact the overall applicability of our encoding.

4.7 Quantifier Landscape

At first glance, the new encoding, except for user-provided quantifiers and domains, seems completely quantifier-free. However, behind the scenes, recursive functions are still encoded using quantifiers by the SMT solver.

Feature	Encoding
Non-Recursive Functions	Quantifier-Free
Recursive Functions[1]	Quantifier-Free encoding
Sequences	Quantifier-Free, except for seq.range[1]
Sets[2]	Quantifier-Free
Multisets	Quantifier based
Maps	Quantifier based
Termination plugin	Quantifier based, except for Int, Bool, Perm & Rational
ADTs	Quantifier-Free
Magic Wands	Quantifier-Free
Impure Assumes	Quantifier-Free
Sort Wrappers	Quantifier-Free
Quantified-Permissions[3]	Quantifier based
Domains[4]	Quantifier based

Conclusion

As we can see, we were able to encode a lot of features quantifier-free. The remaining built-in types and the termination plugin still offer potential for quantifier-free alternatives that were unfortunately outside the scope of this work. The only features that do not seem to have viable quantifier-free alternatives at this moment are recursive functions, Viper domains, and user-provided quantifiers.

[1] Even though the encoding of recursive functions is quantifier-free, in reality, SMT solvers will use quantifiers to encode them under the hood.

[2] Sets are currently only supported in CVC5; for Z3, a domain-based encoding is still in use.

[3] Although quantified permissions naturally involve quantifiers, additional quantifiers are generated that could be encoded as interpreted functions or ADTs.

[4] The current domain implementation also naturally includes domain axioms. However, since domains are still the only option for polymorphic, non-ADT types and functions, the use of quantified axioms could be reduced by offering other mechanisms to encode polymorphic functions or allow interpreted functions in domains.

Chapter 5

Evaluation

In this chapter, we evaluate the encoding presented in chapter 4 against the original Silicon encoding.

Our experimental setup is the following:

Software Versions:

- JDK: OpenJDK 11.0.29
- SBT: 1.9.9
- Original Silicon: v.25.08-release
- Z3: 4.8.7 - 64 bit
- CVC5: 1.3.1 @ea1b484

Shared Silicon Configuration:

- `--numberOfParallelVerifiers 1`: Sets the number of parallel running SMT solvers to one for method verification. This allows for condensed SMT-Solver output and more deterministic results.
- `--timeout 100`: Specifies the upper time limit of 100 seconds that Silicon should take for each test case. After that, verification is aborted, and a timeout is reported.
- `--proverEnableResourceBounds`: Specifies that instead of time limits, the SMT-Solver should use resource limits to determine saturation timeouts and checks for branch feasibility. This allows for more reproducible results.

CVC5 Specific Configuration:

- `--proverResourcesPerMillisecond 200`: The factor of resources per millisecond of time limits that should be used by the solver.
- `(set-option :inst-max-rounds 20)`: Limits the number of quantifier instantiation rounds. If they are exceeded, the solver reports `unknown`. In some

cases, this leads to more complete results, as the SMT-Solver will not get stuck in endless quantifier instantiation rounds.

- `(set-option :sub-cbqi true)`: Specifies that the `cbqi` solver might be used as a sub-solver in quantifiers. This option allowed the verification of a few additional programs when using the `cbqi` heuristic.

Z3 Specific Configuration:

- `--proverResourcesPerMillisecond 1600`: The factor of resources per millisecond of time limits that should be used by the solver.

5.1 Quantifier Heuristics

In this experiment, we want to test how well different quantifier instantiation heuristics perform with our new encoding. We focus on the following heuristics provided by CVC5:

(1) **E-Matching**: As already introduced, e-matching instantiates quantifiers when encountering terms that match a set of defined trigger expressions.

(2) **Model-Based Quantifier Instantiation**: This heuristic operates by using the current model from the SMT solver to generate instantiations that may refute it, with completeness guarantees in certain scenarios. [8]

(3) **Finite-Model-Find**: This technique searches for finite models by bounding the quantifier domains, particularly useful for satisfiable problems where domains are uninterpreted or small. CVC5 also supports finite-model-find for recursive functions, which could lead to more complete results when confronted with recursive functions in certain scenarios [11].

(4) **Conflict-Based Quantifier Instantiation**: Detects when an instantiation is in conflict with the current set of assertions, focusing only on instantiations that guarantee a conflict with the current model for fast, though incomplete, reasoning. [8]

(5) **Counterexample-Guided Quantifier Instantiation**: This technique instantiates quantifiers using a counterexample-guided approach. [12]

For this experiment, we run different combinations of heuristics together and compare the verification results against the original Silicon with its default configuration as an SMT solver. We do not compare heuristic variations in the original Silicon with Z3, since it focuses solely on optimizing the e-matching approach and will therefore likely perform very poorly across different configurations.

5.1.1 Candidate Configurations

We built a list of candidates by combining multiple quantifier-instantiation heuristics. The goal is to make a statement about the heuristics that work best with the new

Viper encoding. Note that the list is far from complete and there are many other possibly interesting combinations and options not included in this experiment. Our goal is not to find the best configuration, but rather to evaluate the effectiveness of our encoding under different heuristics.

Candidate	EM	CBQI	CEGQI	FMF	MBQI	FMFFun
no-heuristics	-	-	-	-	-	-
em	✓	-	-	-	-	-
cbqi	-	✓	-	-	-	-
mbqi	-	-	-	-	✓	-
cegqi	-	-	✓	-	-	-
fmf	-	-	-	✓	-	-
fmf-fun	-	-	-	✓	-	✓
em-cbqi	✓	✓	-	-	-	-
em-cegqi	✓	-	✓	-	-	-
em-fmf	✓	-	-	✓	-	-
em-fmf-fun	✓	-	-	✓	-	✓
em-mbqi	✓	-	-	-	✓	-
fmf-mbqi	-	-	-	✓	✓	-
fmf-cbqi	-	✓	-	✓	-	-
fmf-cegqi	-	-	✓	✓	-	-
fmf-fun-mbqi	-	-	-	✓	✓	✓
fmf-fun-cbqi	-	✓	-	✓	-	✓
fmf-fun-cegqi	-	-	✓	✓	-	✓
mbqi-cbqi	-	✓	-	-	✓	-
mbqi-cegqi	-	-	✓	-	✓	-
cbqi-cegqi	-	✓	✓	-	-	-
no-ematching	-	✓	✓	✓	✓	✓
all-on	✓	✓	✓	✓	✓	✓

Table 5.1: CVC5 quantifier instantiation heuristic configurations. em = E-Matching, cbqi = Conflict-Based QI, cegqi = Counterexample-Guided QI, fmf = Finite Model Find, mbqi = Model-Based QI, fmf-fun = finite model find for recursive functions

5.1.2 Results

We ran all candidates on 176 example files. The test suite only contained features we adapted in the new encoding, specifically excluding quantified permissions and domains. We track for how many examples the candidates give the same result as the original Silicon, which we consider a **Correct result**. In the column **Timeouts**, we track for how many examples the verification timed out. Runs that changed a verification success of the original Silicon into a failure (without timeouts) are tracked in the **Less Complete** column. Lastly, runs that turned a timeout or verification failure into a success are tracked in the **More Complete** column.

For the complete test suite original Silicon reports a success 80 times, a verification

error 96 times, and fails with a timeout once.

Candidate	Correct results	Timeouts	Less Complete	More Complete
no heuristics	148	1	26	1
em	157	15	1	3
cbqi	148	1	26	1
mbqi	147	16	13	0
cegqi	148	1	26	1
fmf	145	6	25	0
fmf-fun	147	13	13	3
em-cbqi	157	15	1	3
em-cegqi	157	15	1	3
em-fmf	**161**	**11**	**1**	**3**
em-fmf-fun	155	18	1	2
em-mbqi	154	11	10	0
fmf-mbqi	147	15	14	0
fmf-cbqi	145	6	25	0
fmf-cegqi	145	6	25	0
fmf-fun-mbqi	**142**	**31**	**2**	**1**
fmf-fun-cbqi	148	13	12	3
fmf-fun-cegqi	147	14	12	3
mbqi-cbqi	147	16	13	0
mbqi-cegqi	147	16	13	0
cbqi-cegqi	148	1	26	1
no e-matching	146	26	3	1
all	151	22	3	1

Table 5.2: Results for our test suite. em = E-Matching, cbqi = Conflict-Based QI, cegqi = Counterexample-Guided QI, fmf = Finite Model Find, mbqi = Model-Based QI, fmf-fun = fmf for recursive functions

From these results, we can conclude that e-matching remains an effective quantifier instantiation heuristic for this test suite. The combination of e-matching with finite-model-find (em-fmf) yields the overall best results, achieving 161 correct results with only 11 timeouts and 1 less complete case. The `fmf-fun` and `mbqi` heuristics both seem to increase the number of timeouts that occur. This becomes especially clear when we look at the `fmf-fun-mbqi` candidate, which performed the worst in terms of the number of correct results and timeouts. Although not all examples containing quantifiers or recursive functions timeout when using `mbqi`, all examples that timeout with `mbqi` contain either quantifiers or recursive functions.

The new theory-based encoding also behaves more resiliently in the absence of quantifier instantiation heuristics. To evaluate this, we compared verification success when disabling quantifier instantiation heuristics entirely. The ”no heuristics” candidate using the new encoding successfully verified 53 test cases, whereas the original Silicon with e-matching disabled succeeded on only 33 test cases. This represents an improvement of 20 additional successfully verified files, demon-

strating increased completeness without quantifier instantiation heuristics.

The rest of this section focuses on analyzing which files' results changed compared to the original Silicon and why. We will first focus on the new incompletenesses we introduced

5.1.3 Less Complete Results & Timeouts

There were nine test cases in total for which the new encoding failed to verify successfully (either timing out or reporting verification errors) across all candidates, despite the original Silicon verifying them successfully. We analyze which features appear in these examples to identify potentially problematic encodings:

Test Case	Features used
combinatorics.vpr	Rec-Funs
sorting_algorithms.vpr	Sequences, Rec-Funs, Quantifiers
list_insert.vpr	Sequences, Wands, Rec-Funs
testTreeWand.vpr	Sequences, Rec-Funs, Quantifiers
pldi16_so_NameComparator_true.vpr	Sequences, Rec-Funs, Quantifiers
pldi16_so_NameComparator_false.vpr	Sequences, Rec-Funs, Quantifiers
testTreeRecursive.vpr	Sequences, Rec-Funs
bdd.vpr	Sets, Rec-Funs, Quantifiers
RingBufferRd.vpr	Sequences

Table 5.3: Test cases that timed out or are incomplete for all candidates in the new encoding.

We can see that most timeouts occur in test cases containing sequences, recursive functions, or quantifiers. Sequences and recursive functions appear in 8 out of 9 examples each.

We tried to add bounds to sequences to check if the files verify for sequences with a maximal length. However, this only worked for `sorting_algorithms.vpr`, which verified with a bound of 5. For all other examples, even reducing the sequence bound down to 2 did not change the timeout to a success or failure.

A notable test case in this list is RingBufferRd.vpr, since it is the only test case that also causes a timeout in the absence of quantifier instantiation heuristics. For this file, we therefore presume that the timeout is most likely linked to CVC5's sequence theory. This assumption is additionally supported by the fact that the file verifies successfully if we set the CVC5 option `strings-fmf`, which enables additional finite model find heuristics for the string theory, to `true`. However, this option causes new timeouts for different examples, making it not a viable default.

The only less complete verification error (compared to the original Silicon) for the `em-fmf` candidate is in datatypes/example_2.vpr file, which uses quantifiers, ADTs, recursive functions, and sequences. In this example, the SMT-Solver gave up after

too many instantiation rounds and reported `unknown` as a result. These results suggest that sequences, as well as recursive functions, still need improvements to truly rival original Silicon's encoding.

5.2 Encoding

In this section, we want to highlight the strengths, weaknesses, and peculiarities of the theory-based encoding that we encountered during the evaluation. Additionally, for encodings identified as problematic by section 5.1.3, we want to propose concrete improvements or different encodings that might improve completeness.

5.2.1 Non-Recursive Functions

Our evaluation has not shown any limitations when using non-recursive functions compared to the original Silicon encoding. Using functions as triggers in quantifiers seems to reliably instantiate the quantifiers as it did in the original Silicon when using e-matching. Since we use interpreted functions on the SMT-LIB level, they are completely quantifier-free and will therefore report `sat` results, as long as no other undecidable logic or constructs are used. Even when using long function chains, as our encoding of the magic wand snapshot function does, it has no impact on verification time.

What we noticed, however, is that it is important to always use the weakest possible function definition keyword on the SMT-LIB level. Using `define-fun-rec` instead of `define-fun` for non-recursive functions might make the SMT-Solver wrongly believe that the function contains recursive definitions and might lead to `unknown` results. A simple example is shown in figure 5.1. CVC5 will report `unknown` for this case, stating `INCOMPLETE QUANTIFIERS` as the reason for the incompleteness. Changing the `define-fun-rec` on line two to a `define-fun`, changes the result to `sat`.

```
1 (set-logic ALL)
2 (define-fun-rec test ((i Int)) Int (* i i))
3 (assert (> (test 10) 0))
4 (check-sat) ; unknown
```

Figure 5.1: Example of a non-recursive function in a `define-fun-rec` leading to an `unknown` result

5.2.2 Recursive Functions

Recursive functions are the only construct we introduced in the new axiomatization that, even if on the surface quantifier-free, depend on quantifiers in CVC5. Using recursive functions will generally yield `unknown` results instead of `sat` results on the SMT-LIB level, except if we apply the `fmf-fun` heuristic in CVC5.

Unrolling

In some cases, our encoding of recursive functions is more complete than that of the original Silicon; in others, it is not. The main reason for both improvements and deterioration in completeness is the unrolling of the recursive function definition. Consider the example in figure 5.2.

```
1 function fac(n: Int): Int
2   requires n >= 0
3   ensures result >= 0
4 { n < 2 ? n : n * fac(n - 1) }

6 method test() {
7   assert fac(3) == 6 // Will fail in original Silicon
8 }
```

Figure 5.2: The assertion needs unrolling to be proven

While in our encoding, the assertion can be correctly verified, in the original Silicon, this is not possible. Recall that original Silicon unrolls recursive definitions lazily, meaning that in this case, only `fac(3)` will be unrolled. However, to actually reason about the value, the SMT-Solver also needs the definition of `fac(2)`. Adding the statement `assert fac(2) == 4` before the other assert causes original Silicon to correctly verify this example as well, since we manually unrolled the definition.

In contrast to this example, the new encoding can run into timeouts because of excessive unrolling. We did not notice an obvious pattern that leads to non-termination. However, a good first step is to make sure the function is actually well-founded, e.g, by proving termination with Viper's termination plugin. Using `fmf-fun` seems to help in some cases, but also makes a lot of examples worse. Technically, it would be possible to lazily unroll the definition in the new encoding as well to check if it achieves better performance.

5.2.3 Sequences

The built-in sequence theory causes many problems in verification that did not occur with the domain axiomatization of the original Silicon. We outline the main problems we encountered during our evaluation. Besides these, we also noticed that the `seq.extract` function can very easily cause verification timeouts.

Triggering

The following example will not verify in the new encoding when depending on e-matching, even though it should for both CVC5 and Z3. The reason for this is that the trigger `i in m` will not instantiate. The same behavior was observed in the previous attempt to use the sequence theory in Silicon. The same example

works when using `mbqi` as a quantifier instantiation, since `mbqi` does not depend on trigger terms to instantiate the quantifier.

```
1  method testSeq(a: Int, m : Seq[Int])
2    requires forall i : Int :: { i in m } i in m ==> i > 0
3    requires a in m
4  {
5    assert a > 0 // fails
6  }
```

Figure 5.3: Sequence example not verifiable using built-in theories

Another workaround we tried was to wrap the `seq.contains` call inside of a custom SMT-LIB function, since triggering seems to work reliably for functions, but to no avail.

Recursive Range definition

The recursive definition of the sequence range function makes verification very slow or unfeasible. To prove properties using the function, the SMT-Solver has to unroll every iteration of the recursive function until a valid model is found, an instantiation limit is reached, or all possible cases have been enumerated. Figure 5.4 shows an example where we construct a range with an unknown upper limit by using an unbounded variable `n`.

```
1  method testSeqRange(n: Int, i: Int)
2    requires i > 0
3  {
4    assert i < n ==> i in [0..n)
5  }
```

Figure 5.4: Example of a non-terminating usage of the sequence range function

While the original Silicon domain encoding handles this case by using smart axioms that encode such behavior, the approach with a recursive function will time out. The example can be verified by bounding `n` to a size, e.g by adding a precondition `n < 10`. However, even when using bounds, the verification will significantly slow down when increasing the bounds.

Alternative sequence encoding

Since we have shown that the sequence encoding of CVC5 and Z3 can run into timeouts even in a quantifier-free context, one could instead try the alternate approach of using SMT-LIB data-types with recursive function definitions. Together with a more careful unrolling of recursive function definitions, this might bring a

more flexible and stable encoding, since we can fine-tune all function specifications ourselves.

5.2.4 Sets

CVC5's set theory functions very well. So far, we have not encountered any performance or completeness deterioration compared to the original Silicon axiomatization. Contrary to our original expectation based on the previous experiences with triggers and sequences, there do not seem to be any issues when using built-in set theory functions inside of triggers, making it more reliable than the sequence theory.

Increased Completeness

We even found that the cardinality function is more complete in CVC5's built-in set theory compared to the domain axiomatization. An example showing this behavior is shown in 5.5

```
1 method testSets(s1: Set[Int], s2: Set[Int])
2 {
3   // Both assertions are currently failing in original silicon
4   assert |s1 intersection s2| <= |s1|
5   assert s1 subset s2 => |s1| <= |s2|
6 }
```

Figure 5.5: Example of a non-terminating usage of the sequence range function

5.2.5 Algebraic Data Types

SMT-LIB's data-type encoding provides an easy and stable way of encoding ADTs and sort-wrappers for Silicon. Besides the problems we outlined in 4.4.1, the only issue we noticed is that when enabling `fmf-fun`, CVC5 seems to run into issues correctly instantiating valid datatypes and frequently reports warnings. Besides the warnings, this does not directly impact verification, so we decided to not further investigate this issue.

5.3 Runtime

In this experiment, we chose the `em-fmf` candidate from section 5.1 and measured its runtime performance against the original Silicon. We only look at the examples where both the candidate and original Silicon do not timeout and report the same result. For this experiment, we repeated each run five times and took the mean runtime of all five runs for comparison.

In figure 5.6, we display the runtime of the `em-fmf` candidate on the x-axis against the runtime of the original Silicon on the y-axis. We can see that for many cases

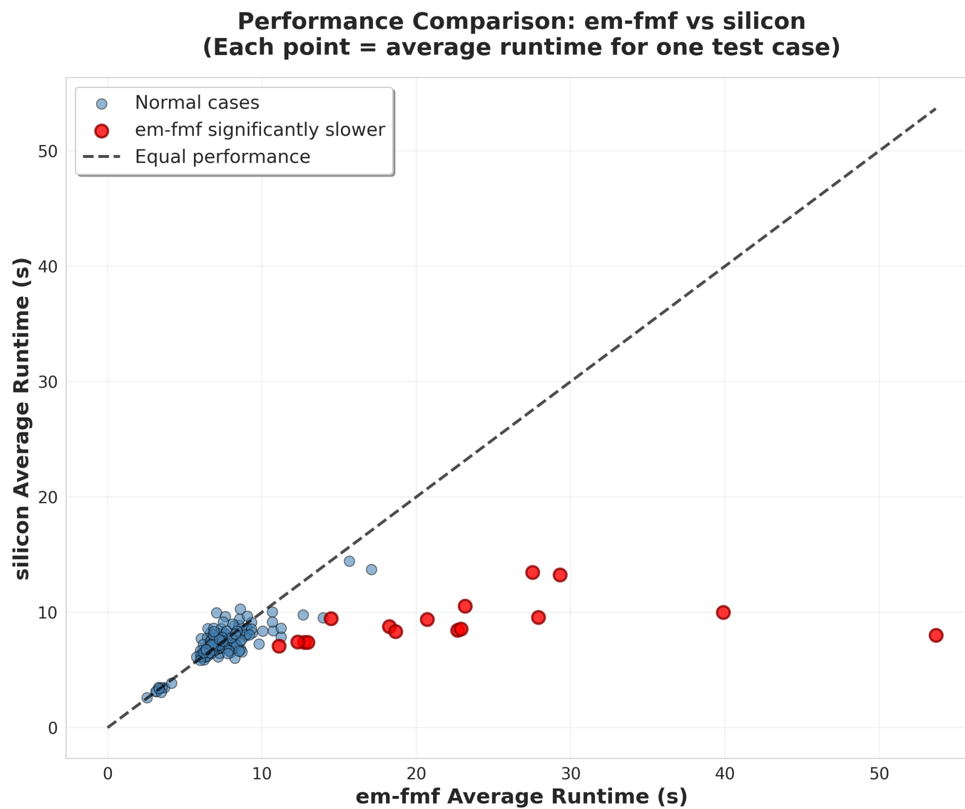

Figure 5.6: Runtime comparison scatter plot: em-fmf candidate (x-axis) vs. original Silicon (y-axis)

`em-fmf` and original Silicon have a very similar runtime. However, there are some cases, for which `em-fmf`'s runtime significantly degrades.

To understand why this happens, we extracted the worst cases (marked in red in figure 5.6) and explored what features they use that might explain their performance degradation. In table 5.4 we list all these outliers with their concrete runtime difference between original Silicon and `em-fmf`.

The results match the pattern we already saw in section 5.1.3. The test cases with the most decreased performance heavily use recursive functions, sequences, and quantifiers. From the runtime analysis, we can therefore conclude that the new encoding can compete with the original Silicon's encoding for many cases. However, the built-in sequence theory and the handling of recursive functions still offer room for future improvement. Nonetheless, these results suggest that a theory-based SMT-LIB encoding is feasible and can offer comparable results to a fully e-matching based solution.

Test Case	em-fmf (s)	orig (s)	Δ (s)	Rel (%)
unofficial003.vpr	53.66	8.02	45.64	568.9
sequence_operations.vpr	39.89	9.99	29.90	299.4
graph_algorithms.vpr	27.91	9.59	18.32	191.1
smith.vpr	29.31	13.26	16.05	121.0
tree_delete_min.vpr	22.89	8.55	14.34	167.7
data_structures.vpr	27.52	13.46	14.06	104.4
list_insert_tmp.vpr	23.15	10.54	12.62	119.7
0286.vpr	18.67	8.36	10.31	123.3
example_3.vpr	18.25	8.76	9.49	108.3
binary_tree_algorithms.vpr	12.96	7.40	5.56	75.2
example_1.vpr	12.74	7.40	5.34	72.3
loop_sum_ghostvar_old.vpr	14.48	9.47	5.01	52.9
SnapshotsLocallyPackaged.vpr	12.32	7.45	4.86	65.2
mathematical_functions.vpr	11.09	7.08	4.02	56.7

Table 5.4: Test cases where em-fmf is significantly slower than the original Silicon

5.4 SMT-Solver Answers

We explored the interaction between Silicon and the SMT-Solver. In the original Silicon encoding, all non-`unsat` responses from the SMT solver are `unknown`, since almost all features involve quantifier reasoning that cannot be decided completely when relying solely on e-matching.

In figure 5.7, we can see the overall responses received by the SMT-Solver for `check-sat` queries. We excluded the files `pldi16_so_NameComparator_true.vpr` and `pldi16_so_NameComparator_false.vpr` from this analysis, since they created a very large number of `check-sat` calls (more than 20000) compared to other test cases. This would have unnecessarily inflated original Silicon's `unknown` responses, since it was the only solver that terminated for this test case. Still, the number of `check-sat` calls varies between each variant, depending on how far they get within the time limit.

As can be seen, the new encoding causes the SMT-Solver to respond with a lot of `sat` answers. The main reasons for `unknown` answers are reached time/resource limits for saturation checks and quantifiers, mainly related to recursive functions and user-provided quantifiers. Depending on the chosen quantifier heuristic, more or fewer `unknown` answers are output. In general, for `mbqi`, very few `unknown` answers are output, since it is a much more complete heuristic than `e-matching`. However, there are also fewer `check-sat` queries, since `mbqi` candidates ran into more timeouts. Another heuristic to note is `fmf-fun`, which yields a much better `sat` response rate, especially when dealing with recursive functions, while similarly to `mbqi`, increasing the risk of timeouts.

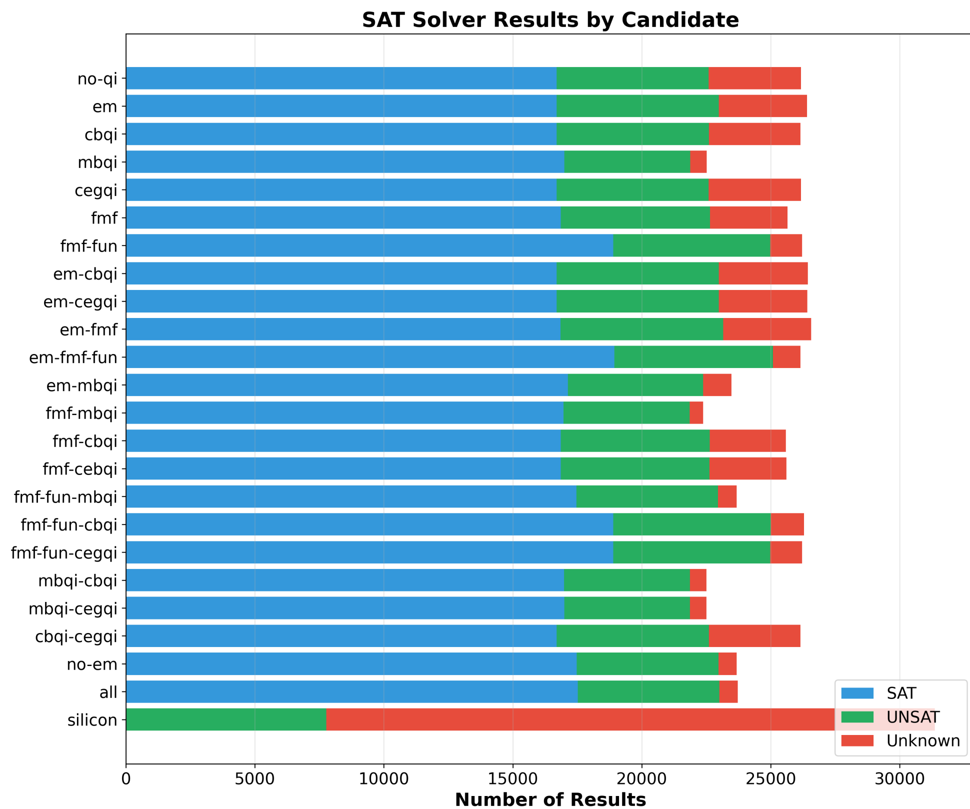

Figure 5.7: Distribution of SMT-Solver responses (sat, unsat, unknown) for check-sat queries across different candidates

In figure 5.8, we can see the total number of test cases for which the SMT-Solver never responded with `unknown`. In practice, we could guarantee, given our encoding is correct, that verification errors for files without `unknown` responses are actual verification errors and not potentially incomplete results. This graph clearly shows that, at least for a subset of Viper, a more complete verification is possible and feasible.

These results could also motivate providing configuration options in the future that use complete quantifier heuristics such as `mbqi` and `fmf-fun` to achieve more complete verification results. Users could then use these configuration options to confirm with high confidence that a reported verification error is a genuine error and not an artifact of incomplete instantiation.

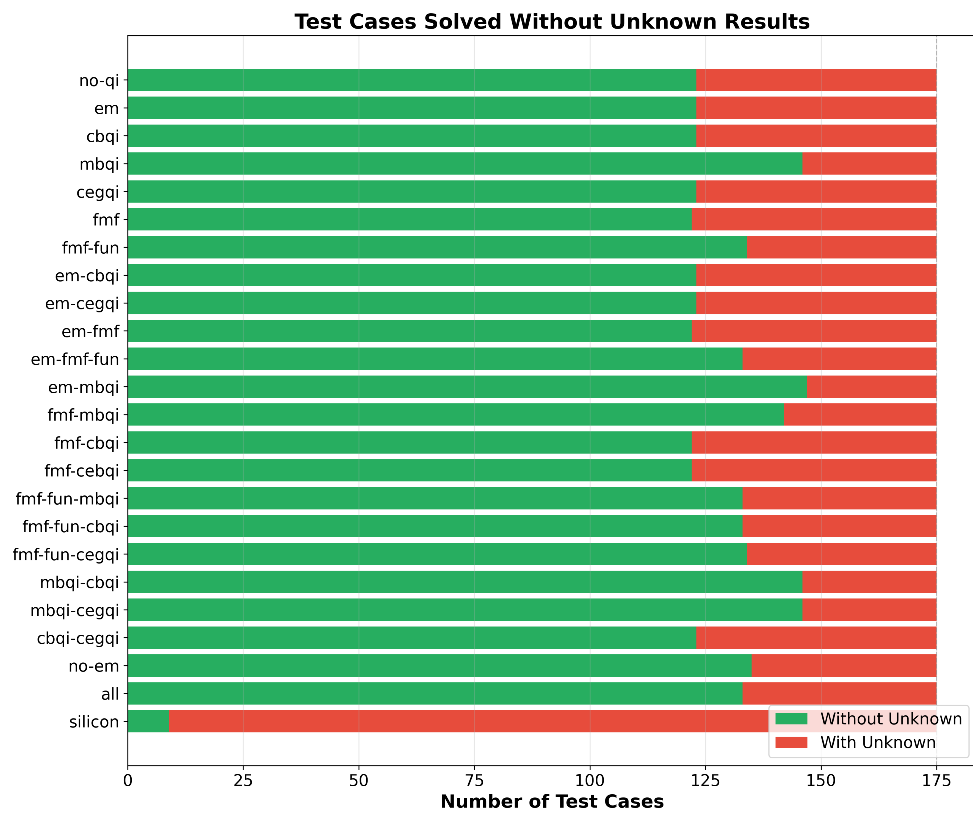

Figure 5.8: Number of test cases for which the SMT-Solver never responded with unknown

Chapter 6

Conclusion

This thesis investigated the feasibility of using an SMT-LIB encoding based on interpreted functions and native SMT theories instead of the traditional approach relying on uninterpreted functions and quantified axioms with e-matching. Through a systematic analysis of Silicon's current encoding, the development of theory-based alternatives, and a comprehensive evaluation, we have demonstrated that such an approach is not only feasible but offers meaningful advantages for substantial subsets of Viper programs.

6.1 Summary of Contributions

We began by documenting the extensive use of quantifiers in Silicon's current encoding (Chapter 3). Our analysis revealed that Viper's language features, such as functions, sort wrappers, domains, magic wands, and more, are predominantly encoded as uninterpreted functions whose semantics are defined through universally quantified axioms. This creates a verification landscape that critically depends on trigger-based instantiation through e-matching, inheriting all of its well-known limitations: incompleteness from restrictive triggers, performance degradation from overly liberal triggers, and unpredictability from subtle changes in term appearance.

Building on this analysis, we developed a comprehensive alternative encoding (Chapter 4) that systematically replaces quantified axiomatizations with quantifier-free or quantifier-reduced alternatives:

- **Functions**: We encoded Viper functions as uninterpreted functions paired with recursive interpreted boolean functions that capture both specifications and definitions. By manually instantiating these definitional functions at call sites, we gain explicit control over instantiation, avoiding e-matching unpredictability.

- **Sort Wrappers**: We replaced quantified injectivity and inverse axioms with algebraic data type constructors and destructors, leveraging the SMT-LIB semantics that automatically provide these properties without quantifiers.
- **Sequences and Sets**: We utilized CVC5's native theories for sequences and sets instead of domain-based axiomatizations, eliminating the need for quantified axioms to define operations like concatenation, membership, and cardinality.
- **Algebraic Data Types**: We translated Viper ADTs to SMT-LIB's native `declare-datatypes` construct, replacing quantified injectivity, exhaustiveness, and discriminator axioms with built-in data type semantics.
- **Magic Wands**: We implemented a quantifier-free encoding through chained function definitions that incrementally extend the magic wand snapshot function as new wands are packaged during program execution.
- **Termination Plugin**: For primitive types (integers, booleans, permissions), we replaced domain axiomatizations of well-founded orders with interpreted Viper functions.

Our evaluation (Chapter 5) demonstrated the practical viability of this approach across 176 test cases from Silicon's test suite. The key findings include:

(1) **Comparable Completeness**: The best configuration (e-matching combined with finite-model-finding) achieved 164 correct results, of which 3 were more complete than Silicon's results, while failing in only 12 results.

(2) **Increased Robustness**: Without any quantifier instantiation heuristics, the new encoding reports 53 verification successes compared to only 33 for the original Silicon, a 60% improvement. This demonstrates that the theory-based encoding is less reliant on quantifier instantiation.

(3) **Increased Completeness for Verification Errors**: The new encoding produces significantly more `sat` responses instead of `unknown` responses. For many test cases, the SMT solver never reports `unknown`, providing high confidence that reported verification errors are genuine rather than artifacts of incomplete instantiation.

(4) **Competitive Performance**: For most test cases, runtime performance is comparable to the original Silicon. However, programs heavily using recursive functions and sequence operations show performance degradation, indicating areas requiring further optimization.

(5) **Flexibility in Instantiation Strategies**: The theory-based encoding works well with various quantifier instantiation heuristics beyond e-matching, including model-based instantiation (MBQI), and finite-model-finding (FMF). This flexibility enables users to trade off completeness and performance based on their verification needs.

6.2 Feasibility of a theory-based approach

Returning to the initial question posed in the introduction, ”*Is it feasible to reduce or eliminate dependence on e-matching by using an SMT-LIB encoding based on interpreted functions and native SMT theories?*”, our work provides a qualified affirmative answer.

For a substantial subset of Viper's features, the theory-based encoding successfully eliminates or significantly reduces quantifier dependence. Non-recursive functions, sort wrappers, algebraic data types, sets, magic wands, and several other features can be encoded in a truly quantifier-free manner. The evaluation demonstrates that this encoding achieves comparable or better completeness than the original axiomatization while offering greater decidability and robustness.

However, important limitations remain. Recursive functions, while encoded using `define-fun-rec` instead of quantified axioms, still involve quantifier-based reasoning internally within CVC5. The built-in sequence theory, while eliminating quantified axioms, introduces new challenges related to triggering and performance. User-provided quantifiers and domain axioms naturally remain quantified, though they interact well with the theory-based encoding in most cases.

The answer is therefore nuanced: the theory-based encoding is feasible and beneficial for large portions of Viper programs, offering improved decidability and robustness. For programs that avoid or minimize recursive functions and complex sequence operations, the encoding can verify with significantly fewer or even no `unknown` results, enabling higher confidence in verification outcomes. For programs that heavily utilize these features, the encoding remains viable but requires further optimization to match Silicon's original performance.

6.3 Future Work

Although this thesis demonstrates the feasibility of theory-based SMT-LIB encodings, several promising directions for future work remain.

6.3.1 Improved Recursive Function Handling

Recursive functions remain a challenge in the current encoding. Two specific improvements could address the limitations identified in our evaluation:

- **Lazy unrolling**: The current encoding eagerly includes nested function definitions in the definitional function, which can lead to excessive unrolling and timeouts. Implementing lazy unrolling, like it is currently done in the original Silicon, could reduce this overhead while maintaining completeness. This would require careful handling of predicate triggers.

- **Termination-guided unrolling**: For recursive functions with proven termination via Viper's termination plugin, the termination measure could guide unrolling strategies.
- **Better integration with FMF-fun**: The finite-model-find heuristic for recursive functions (`fmf-fun`) shows promise for improved completeness but currently causes timeouts in many cases. Investigating why timeouts occur could yield significant benefits.

6.3.2 Alternative Sequence Encoding

The evaluation revealed that CVC5's built-in sequence theory has triggering issues when using e-matching and performance issues that did not occur with the original domain axiomatization. An alternative approach could be an `ADT` based implementation. Instead of using the built-in sequence theory, sequences could be encoded as algebraic data types with recursive function definitions for operations. This would provide more control over triggering and instantiation while potentially avoiding the theory-specific issues we encountered.

6.3.3 Support for Additional Features

Several Viper features were deliberately excluded from this work, but could benefit from theory-based encodings:

- **Multisets and Maps**: These built-in data types remain encoded using domains and quantified axioms. CVC5 provides some support for these theories that could be leveraged, though the support is less mature than for sequences and sets.
- **Quantified Permissions**: Quantified permissions generate additional quantifiers beyond those written by users. Although these quantifiers are inherent to the feature, some of the auxiliary functions and axioms they generate could potentially be encoded as interpreted functions or data types.
- **Polymorphic Functions in Domains**: The current implementation still relies on domains for polymorphic types (other than ADTs) and functions. Extending Viper to support polymorphic interpreted functions would enable more features to benefit from the theory-based encoding.
- **Termination Plugin for Polymorphic Types**: The termination plugin's well-founded orders for sequences, sets, and other polymorphic types remain axiomatized. Developing interpreted function encodings for these types would make termination verification, since they are generally non-recursive, quantifier-free, and therefore more complete.

6.3.4 Solver Integration and Configuration

The evaluation demonstrated that different quantifier instantiation heuristics offer different trade-offs between completeness and performance. Future work could explore:

- **Adaptive heuristic selection**: Automatically selecting appropriate instantiation strategies based on program characteristics.
- **User-configurable verification modes**: Providing preset configurations optimizing for different goals: fast verification with potential incompleteness, complete verification with higher timeouts, or maximum completeness for bug confirmation.
- **Z3 support optimization**: While our encoding targets CVC5, broader support for Z3 would be valuable. The set encoding currently falls back to domains for Z3; investigating Z3's array-based set representation or alternative encodings could enable full theory-based verification with Z3.

6.3.5 Further Tests with Viper Frontends

This thesis focused on feasibility and performance evaluation using Viper programs from Silicon's test suite, which are primarily small, focused examples designed to test specific language features. While this approach was appropriate for establishing the viability of the theory-based encoding, it leaves an important question unanswered: how does the encoding perform on real-world programs verified through Viper's frontends?

The programs generated by these frontends often exhibit specific patterns and characteristics, different from handwritten Viper test cases. They also tend to be larger, use features in combinations not commonly seen in isolated tests, and may generate encoding patterns optimized for their specific source language semantics.

Several questions warrant investigation through frontend-based evaluation:

- **Completeness in practice**: The evaluation showed that certain instantiation heuristics provide better completeness for specific feature combinations. Do real-world programs cluster around feature combinations where the theory-based encoding excels, or do they frequently require the few feature combinations where the original encoding performs better?
- **Decidability benefits**: One of the key advantages demonstrated was increased decidability, more `sat` results, and fewer `unknown` responses. For developers using Viper frontends, this could translate to higher confidence that reported errors are genuine bugs rather than verification artifacts. Quantifying this benefit in real programs would provide valuable evidence of the practical impact of the encoding.

- **Frontend-specific encoding patterns**: Each frontend makes specific choices about how to encode source language features into Viper. Understanding whether certain frontend encoding strategies interact particularly well or poorly with the theory-based encoding could guide both frontend and backend optimization efforts.

A comprehensive evaluation using suites of verified programs from each major frontend would provide crucial evidence about the practical applicability of this work. Such an evaluation would also help identify which types of verification tasks benefit most from the theory-based encoding, potentially leading to hybrid approaches where encoding strategies are selected based on program characteristics.

6.4 Final Remarks

This thesis demonstrates that modern SMT solvers' built-in theories and constructs enable verification encodings that are less dependent on the heuristics and unpredictability of e-matching. By systematically replacing quantified axioms with interpreted functions and native theories, we achieved an encoding that produces more complete verification error results, exhibits greater robustness without quantifier instantiation, and maintains competitive performance for most Viper programs.

Although challenges remain, particularly for recursive functions and sequence operations, the results strongly suggest that theory-based encodings represent a feasible goal for SMT-based program verification. As SMT solvers continue to evolve and expand their theory support, the gap between axiomatized and theory-based approaches is likely to widen further in favor of the latter.

We hope this work encourages further revisiting of encoding strategies developed for earlier generations of SMT solvers and the exploration of how modern solver capabilities can improve verification completeness and decidability.

Bibliography

[1] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[2] M. Eilers and P. Müller, *Nagini: A Static Verifier for Python: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, 07 2018, pp. 596–603.

[3] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification (CAV)*, A. Silva and K. R. M. Leino, Eds. Springer International Publishing, 2021, pp. 367–379.

[4] M. H. Schwerhoff, "Advancing automated, permission-based program verification using symbolic execution," Ph.D. dissertation, ETH Zurich, 2016.

[5] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[6] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 415–442.

[7] C. Barrett, P. Fontaine, and C. Tinelli, “The SMT-LIB Standard: Version 2.7,” Department of Computer Science, The University of Iowa, Tech. Rep., 2025.

[8] A. Reynolds, “Conflicts, models and heuristics for quantifier instantiation in smt,” in *Vampire@IJCAR*, 2017.

[9] S. Heule, I. T. Kassios, P. Müller, and A. J. Summers, “Verification condition generation for permission logics with abstract predicates and abstraction functions,” in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 451–476.

[10] L. Schär, “Supporting sequence axiomatization on the smt solver level for the viper project,” Bachelor’s Thesis, Department of Computer Science, ETH Zurich, 2016.

[11] A. Reynolds, J. C. Blanchette, S. Cruanes, and C. Tinelli, “Model finding for recursive functions in smt,” in *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 133–151.

[12] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, “Counterexample-guided quantifier instantiation for synthesis in smt,” in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 198–216.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden zwei Optionen ist **in Absprache mit der verantwortlichen Betreuungsperson** verbindlich auszuwählen:

☐ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.

☒ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe. Dabei habe ich nur die erlaubten Hilfsmittel verwendet, darunter sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson sowie Technologien der generativen künstlichen Intelligenz. Deren Einsatz und Kennzeichnung ist mit der Betreuungsperson abgesprochen.

Titel der Arbeit:

Theory-Based Verification Condition Encoding

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Gasser	Johannes

Ich bestätige mit meiner Unterschrift:

- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Muttenz, 08.12.2025	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] Für weitere Informationen konsultieren Sie bitte die Webseiten der ETH Zürich, bspw. https://ethz.ch/de/die-eth-zuerich/lehre/ai-in-education.html und https://library.ethz.ch/forschen-und-publizieren/Wissenschaftliches-Schreiben-an-der-ETH-Zuerich.html (Änderungen vorbehalten).