Symbolic Execution of GRASShopper Programs

by

Eric Cox

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

New York University

December, 2021

Professor Thomas Wies

To Kitty and James.

Acknowledgments

This work wouldn't be possible without my thesis advisor, Thomas Wies. I say this because without his patience and continual encouragement I would have probably dropped out of my Masters. Whenever I felt like I was at the limits of my understanding or I've hit a seemingly dead end our discussions made details clear and his advice gave me the courage to carry forward. I looked forward to and will miss our weekly meetings. I'd also like to thank Siddharth Krishna for initially on-boarding me onto the GRASShopper codebase and answering my questions about formal verification when I was first starting.

As always, I'd like to thank my wife Kitty for being supportive and encouraging of this endeavor as I couldn't have done it without her support and companionship while I was typing away. I would have never attempted my return to Graduate studies if it wasn't for my son, James. He made me laugh when I was tired and helped me realize that I should solidify my career transition by returning to school so I can support him and provide him stability.

Lastly, I'd like to thank Ethan Zhang, James Whalin, and Pawel Terlecki at MongoDB for supporting my decision to finish my studies while slinging code and helping me grow my career before I graduated.

Abstract

Symbolic execution is a more efficient and viable alternative to implementing deductive verification tools to fully automate the formal verification of programs. Symbolic execution in many cases can provide performance gains over *verification condition generation* (VCG) based verification tools due to the fact that symbolic execution directly manipulates in-memory data structures.

This thesis presents the design and implementation of a symbolic execution engine for the GRASShopper programming language which already supports verification using VCG. The goal of this work was to adapt ideas from the symbolic execution engine from the Viper verification infrastructure language to the semantics of GRASShopper and to demonstrate its utility on sample programs. We present a rigorous description of the operational semantics of the symbolic interpreter, discuss implementation details and illustrate the symbolic execution behavior on a set of sample programs.

In order to explore interesting details around implementing a symbolic execution backend for GRASShopper, this work introduced a method to support encoding of snapshots at the struct field level using injective functions. In addition, several language extensions were added to the GRASShopper user-facing language and the intermediate representation. A few of these extensions will now allow support for finer-grained permissions for individual fields rather than granting permissions to all fields of structures, unfolding and folding of recursive predicates, and support for if-then-else expressions in predicate and heap-dependent functions.

Contents

Dedication iii

Acknowledgments iv

Abstract v

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Background and Related Work 2

1.1.1 Implementation of Deductive Verification Tools 3

1.1.2 Separation Logic 4

1.2 Contributions 5

2 Preliminaries 7

2.1 Syntax 7

2.2 Language extensions 11

2.3 Notation 12

2.4 Verification as an SMT Problem 13

3 Design & Theory **15**

3.1 Semantic Domains . . . 15

3.2 Operational Semantics . . . 17

3.2.1 Execution Rules . . . 18

3.2.2 Produce & Consume Rules . . . 20

3.2.3 Symbolic Expression Evaluation Rules . . . 22

3.3 Snapshot Encoding . . . 24

3.3.1 Injective Axioms . . . 26

3.3.2 Snapshots as ADTs . . . 28

3.4 State Consolidation . . . 29

3.5 Well-Formedness and Validity Checks . . . 30

3.6 Predicates . . . 33

3.7 Heap-dependent Functions . . . 34

3.7.1 Axiomatization . . . 35

3.8 Unfolding Axiom Generation . . . 37

4 Implementation & Experiments **40**

4.1 Linked List Data Structures . . . 40

4.2 Tree Data Structures . . . 42

4.3 Results . . . 44

5 Conclusion **47**

A Appendix **48**

A.1 Sample Code for Evaluation Results . . . 48

Bibliography **59**

List of Figures

2.1 Example of the user-facing GRASShopper program to illustrate the translation to the IR. . . . 7

2.2 Subset of the GRASShopper IR for terms and formulas . . . 9

2.3 Subset of the GRASShopper IR for loop-free commands . . . 10

3.1 Semantic domains of the GRASShopper symbolic execution semantics . . . 16

3.2 Execution rules for the GRASShopper Cmd domain. . . . 19

3.3 A subset of produce rules for the GRASShopper Formula domain. . . . 20

3.4 A subset of the consume rules for the GRASShopper Formula domain. . . . 21

3.5 A subset of the symbolic expression rules for the GRASShopper Terms domain. . 23

3.6 A counter predicate definition and a procedure that increments the counter by one The `ghost` variable `v` is solely used to verify that the counter is incremented correctly. The predicate `counter` gives permissions to access the `c.count` member of the `Counter` and it enforces that the counter's `count` is equal to `v`. The `counter` predicate can be used in the precondition of `incr` and folded and unfolded in the body. . . . 25

3.7 Symbolic state during verification of `incr`. . . . 26

3.8 A counter client program with annotations of the symbolic state to to demonstrate framing. . . . 27

3.9 An example of incompleteness due to aliasing that can be resolved by state consolidation. . . . 29
3.10 An example of incompleteness due to aliasing that cannot be resolved by state consolidation. . . . 30
3.11 A function to infer disequalities from aliased heap chunks. . . . 30
3.12 Well-formedness and validity checking rules for procedures, functions, and predicates. . . . 32
3.13 Definition of a node for a singly-linked list and an `lseg` predicate. . . . 34
3.14 A heap-dependent function that adds data from two `Node`s in a client procedure. . . . 35
3.15 Definition of a length function that leverages `unfolding` an `lseg` predicate. For a definition of `lseg` see Figure 3.13. . . . 37

4.1 A procedure to remove the first node of a linked list with annotations for the symbolic state during the execution. . . . 41
4.2 Procedure to append a node to the end of a linked list. . . . 42
4.3 Procedure to reverse a linked list with an iterative algorithm. . . . 43
4.4 Definition of a node for a binary tree and a tree predicate. . . . 44

A.1 Basic assignment code. . . . 48
A.2 Basic loop code. . . . 49
A.3 Basic pure assertion code. . . . 50
A.4 Basic if-then-else and `old` expression code. . . . 51
A.5 Code to illustrate framing of procedure calls. . . . 52
A.6 Code to illustrate heap-dependent function calls. . . . 53
A.7 Code to illustrate fold/unfold verification primitives. . . . 54
A.8 Code for appending a node to the end of a linked list 55
A.9 Code for removing a node from a linked list and reversing a list. . . . 56

A.10 A simple procedure that will recursively traverse a tree. 57
A.11 Insertion of a value into a tree-set. 57
A.12 A left and right rotation of a tree-set. 58

List of Tables

4.1 Evaluation table for GRASShopper programs checked by symbolic execution. The data was collected on a Linux machine running an Intel Core i7-8565U CPU with 1.80GHz 4 core processor, 32 GB of ram, and the OCaml compiler at version 4.08.1. All source code is listed in Appendix A and LoC denotes lines of code. 45

1 Introduction

Would you buy a house or work in a skyscraper that wasn't constructed by following a well-designed blueprint? A reasonable person would answer *no*. Unfortunately, the same perception isn't applied to the ubiquitous use of software in today's society. The truth is that the majority of large software systems are never constructed from such a blueprint (e.g., a specification) and most engineers that build these systems only rely on extensive unit-testing to demonstrate that the test cases match some expected behavior. This is rather limited since all that this proves is that there are no bugs for the set of program traces that the unit-tests target. What would be better is a proof that says a given system is bug-free. Regardless of this issue, humans have placed their trust in large tech companies to store their most personal information and the memories of our lives. Unverified software systems manage our finances, control our vehicle's braking and acceleration systems.

Software defects are inevitable. Experiencing an iPhone crash or Gmail outage may be an expensive inconvenience, but recall the Therac-25 radiation therapy device that was involved in six incidents between 1985 and 1987 where cancer patients were given massive overdoses in radiation resulting in death. The incident was tied back to defects in software controlling the device. These incidents aren't just a thing of the past. Tesla had to recall 11,706 vehicles over a full self-driving beta software bug that caused phantom breaking. In addition to these catastrophic failures, we hear about software defects causing data-center outages which lead to loss of business and disruptions for the systems that depend on these services. Software developers

also spend a large amount of time dealing with bug reports, server crashes and back-traces sent from customers.

This makes one wonder if there's a better way to construct software to reduce the chance of experiencing catastrophic defects. Thankfully, there is a promising trend that since the early 2000s where *deductive verification tools* have begun to be developed and adopted by industry at scale to prove that software meets a specification, and when it's proven that the software meets a given specification certain bugs such as buffer overflows don't exist. These deductive verification systems aim to formally verify that software meets a given specification, and in turn a formally verified system will have proven that defects are absent.

1.1 Background and Related Work

Deductive software verification has had a long history, beginning in the late 1940s with an early paper by Turing [27] on verification that introduces some of the ideas that are later formulated in Floyd-Hoare Logic [11, 8] and followed by Dijkstra's weakest precondition [7]. The latter are currently used in automated deductive verification tools such as GRASShopper [21] and Dafny [17]. A good overview of the history of deductive verification methods is given by Hähnle and Huisman [10]. In this work, they mention that the idea of using symbolic execution (SE) to prove program correctness was introduced by Burstall [4]. In this approach a program is executed symbolically with induction to prove that a program implies its post-condition. These seminal works enabled several tools that leverage Verification Condition Generation (VCG) and SE [10]. Notable deductive verification that implement VCG are SPARK [5], Dafny, and Viper (the verification language that inspired this work). The language SPARK began in an academic setting but was later extended for usage in an industrial setting and is actively maintained by AdaCore and Altran [10]. Dafny was developed recently by the RiSE group at Microsoft Research [17] and Viper [18] uses both VCG and SE. Besides VCG-based systems KIV is a notable one that is based on dynamic

logic [9]. The KIV verifier was the first to formalize Burstall's approach as a symbolic interpreter and offered an alternative to VCG-backed tools.

1.1.1 Implementation of Deductive Verification Tools

In general, there are two main approaches to developing deductive verification tools: VCG and SE. A VCG-based tool will essentially apply transformations to the program to reduce the program into a set of verification conditions that will imply the post-condition. After the VCG formulas are generated, the correctness of the program encoded in the verification conditions will be checked by a satisfibly modulo theories (SMT) solver. Both Dafny and Viper implement this VCG transformation by lowering their respective source programs to the Boogie [15] intermediate representation. The Boogie language can then be used in specialized VCG algorithms to generate the final set of constraints that are checked by the SMT solver to prove correctness of a program.

Symbolic execution is a fundamentally different approach and a viable alternative approach to VCG. SE is essentially implemented as an interpreter that will execute the program on an abstract machine where program variables are replaced by uninterpreted symbolic values that can take on any value. These uninterpreted values can be instantiated by the SMT solver in their domain which in-turn will represent all possible values a program variable may take during the actual execution of the program. The configuration of the interpreter is designed to accumulate constraints to represent the possible traces of the program execution and the constraints on symbolic values. These constraints can be be used to check entailment of the post-condition of a program to verify correctness with an SMT solver.

VCG and SE have very different performance characteristics even though both methods will discharge queries to an SMT solver. The performance of a given method depends on the characteristics of the program being analyzed. In general, because SE is mutating in-memory data structures in the symbolic interpreter's configuration and is a more fined grained checking process, SE will usually outperform VCG. However, if a program has a lot of branching, the branch

conditions in the SE state may become large and SE could have worse performance than VCG. A study done by Kassios et al. [14] compared the performance of VCG-based Chalice [16] to Syxc, which is an extension for Chalice using a symbolic execution backend [23]. This study showed that verification based on symbolic execution is roughly twice as fast as verification by VCG. Some notable examples of deductive verification tools that support both VCG and SE are VeriCool [25], Chalice, and Viper.

1.1.2 Separation Logic

The GRASShopper language supports proofs based on *separation logic* (SL) [22, 19, 20] to reason about programs that manipulate the heap. The main goal of this work was to build an SE engine as an alternative to using VCG for checking GRASShopper programs that use SL for their specifications. Thus we will briefly introduce the idea behind SL and highlight some other tools that use SE to support SL.

At its core, SL is an extension of Hoare logic [11] to support modular reasoning over memory regions using in-place updating of logical facts. These logical facts are structured in a way to mimic the way a machine would update its internal state. For example, if one was to use SL to prove that an assignment to a field $\mathtt{x.f} := 3$ is safe and preserves the value of another disjoint heap region $\mathtt{y.f}$ one would write a Hoare triple of the form,

$$\{\mathtt{x.f} \mapsto v * \mathtt{y.f} \mapsto w\}\ \mathtt{x.f} := 3\ \{\mathtt{x.f} \mapsto 3 * \mathtt{y.f} \mapsto w\}. \tag{1.1}$$

SL introduces a *points-to* relation, $\mathtt{x.f} \mapsto v$ which allows permissions to a variable and states a constraint such as $\mathtt{x.f} == v$ on the region's location. To support modular reasoning and the idea that assertions $\mathtt{x.f} \mapsto v$ and $\mathtt{y.f} \mapsto w$ can be separately combined, SL introduces a *separating conjunction* operator, or a *-conjunct to combine permissions [19]. Now the crux of the proof for the Hoare triple in Equation 1.1 is to decompose the triple into a local proof of the actual update

on x.f as,

$$\{\mathtt{x.f} \mapsto v\}\ \mathtt{x.f} := 3\ \{\mathtt{x.f} \mapsto 3\}. \tag{1.2}$$

Using the *frame rule* from SL, $\{P\}c\{Q\} \implies \{P * R\}c\{Q * R\}$ [19], we can *frame* the triple in Equation (1.2) with a heap region $R \triangleq \mathtt{y.f} \mapsto w$ to obtain (1.2). The logic, thus, guarantees for free (i.e. via simple syntactic manipulation of assertions) that the disjoint region R is not affected when x.f := 3 is executed. In SL the Hoare triple guarantees memory safety (no access to unallocated memory), and the in-place reasoning that enables local proofs to be combined into proofs over larger memory regions is useful for proving properties of heap-dependent data structures.

The first tool created to automate the checking of SL based programs was Smallfoot [3] which was based on SE rather than VCG and uses the idea of maintaining permissions on a symbolic heap separately from stacks of constraints over values. The Viper project's SE engine, Silicon [18], is builds on and extends the ideas first explored in Smallfoot. Other notable tools that implement SE engines for variants of SL are Verifast [12] and Chalice [16]. Viper, Verifast, Chalice, and GRASShopper all consider a variant of SL called *implicit dynamic frames* [25].

Implicit dynamic frames (IDF) uses access predicates `acc(x.f)` to state permissions on a memory region $x.f$, and can support heap-dependent constraints such as `x.f == 10` within SL statements. For example, a logical statement over a heap region x.f that the field's value is 10 written in separation logic would be $\exists z.\, \mathtt{x.f} \mapsto \mathtt{z} \wedge \mathtt{z} == 10$. IDF would shorten this assertion to say `acc(x.f) &*& x.f == 10` which eliminates the need for the intermediate variable z.

1.2 Contributions

The main goal of this thesis was to implement a SE verification backend for the GRASShopper language. Much of the design and implementation was inspired by the Viper SE engine, Silicon. The main motivation for doing this was that GRASShopper only supported VCG and Prof Wies'

group was interested in extending GRASShopper to support SE of Iris-style higher-order separation logic [13]. So this work was an investigation on how to build an SE backend based on the Viper Silicon engine so we can later extend the SE backend to support Iris. The latter wasn't in scope for this thesis, but the knowledge and lessons learned by this project can be carried over.

Like Viper, GRASShopper is based on IDF, this seemed like a natural model to follow. However, given this fact, this work wasn't a direct port of the SE backed from Viper to GRASShopper since the semantics of GRASShopper differed from Viper in a few ways. One example where the semantics differed is that Viper only has a single reference type whereas GRASShopper implements pointers to user-defined struct types. This means that Viper's snapshot mechanism for framing needed to be extended to support arbitrary types. Viper was also designed to target the verification of garbage collected languages such as Java, whereas GRASShopper does not assume garbage collection and instead relies on manual memory management. In addition to adapting the Viper SE engine to GRASShopper, this project improved upon the implementation of Viper's `join` rule for evaluating if-then-else expressions, and unfolding expressions. We also provide a more thorough description of how the `join` rule relates to the axiomatization of heap-dependent functions.

2 Preliminaries

2.1 Syntax

The GRASShopper symbolic execution engine was designed to symbolically execute the GRASShopper intermediate representation (IR) rather than the user-facing syntax. Before presenting the syntax of the IR, we will briefly illustrate a few examples of the user-facing GRASShopper syntax and demonstrate the translated code in the IR. Consider a program that writes to a memory region containing a struct value in Figure 2.1.

```
1 struct Node {
2   var next: Node;
3 }
4
5 procedure lookup_heap_assign(x: Node) returns (y: Node)
6   requires acc(x.next) &*& x.next == null
7   ensures acc(x.next) &*& acc(y.next) &*& x.next == y
8 {
9  y := new Node();
10  x.next := y;
11 }
```

Figure 2.1: Example of the user-facing GRASShopper program to illustrate the translation to the IR.

The program in Figure 2.1 allocates a new `Node` and then assigns it to `x.next`. The specification contains a pre-condition, line 6, that requires access permission to the field `x.next` and a constraint that the value of `x.next` is equal to `null`. The post-condition in line 7 states that when the procedure `lookup_heap_assign` returns, permissions to `x.next` and `y.next` are given back to

the caller and `x.next == y` is a true statement.

Since the SE engine for GRASShopper targets the IR, we will now discuss how the program in Figure 2.1 is represented in the IR. The program declares a struct type `Node` that only consists of a field `next` which will have a type `Loc⟨Node⟩`. The specification of `lookup_heap_assign` begins with a pre- and post-condition in lines 6, and 7, respectively. The pre-condition,

```
acc(x.next) &*& x.next == null
```

is a separation logic formula and results in the following code,

```
acc(x.next) &*& (read(x, next) == null).
```

The `read(x, next)` syntax describes the field read expression. Similarly, the post condition is also a separation logic formula and is thus translated to

```
(acc(x.next) &*& acc(y.next)) &*& (read(x, next) == y).
```

The body of `lookup_heap_assign` in lines 9 and 10 of Figure 2.1 is a classical sequence control between two commands C_1; C_2 and is thus a sequence of

```
y := new Node(); next := write(next, x, y).
```

One can see that the IR is simpler than the user-facing syntax and we chose to implement the SE engine over the simpler IR rather than the user-facing syntax.

We now present the syntax of the IR in order to define the SE rules of GRASShopper for the syntactic domains. The syntax definitions are divided into two parts; one for terms and formulas, and the other for loop-free commands. We note that the front-end of GRASShopper translates loops into tail-recursive functions, so loops are omitted from the IR. The sub-grammar for terms

$i \in \mathsf{Int}$		Integers, $\mathbb{Z}$
$id \in \mathsf{Ident}$		Identifiers
$\tau \in \mathsf{Terms} ::=$	$\mathsf{x} \mid \mathsf{s}(\overline{\tau})$	Terms
$\mathsf{s} \in \mathsf{Symb}$		Symbols
$\mathsf{s} ::=$	$- \mid + \mid / \mid \% \mid \ldots$	Arithmetic symbols
	$\mid$ `null` $\mid$ `read` $\mid$ `write`	Uninterpreted location symbols
	$\mid$ `&&` $\mid \| \mid \neg \mid$ `ite`	If-then-else symbols
	$\mid == \mid \geq \mid \leq \mid < \mid >$	Boolean predicate symbols
	$\mid$ `old`	Oldification symbol
	$\mid id$	Uninterpreted identifier
	$\mid$ `true` $\mid$ `false` $\mid i$	Boolean and integer constants
	$\mid$ `unfolding`	Unfolding symbol
$f \in \mathsf{Form}$		Formulas
	$\mid \mathsf{bop}(\overline{f})$	Boolean Operations
	$\mid \forall \overline{x} \in \overline{T}. f(\overline{x}) \mid \exists \overline{x} \in \overline{T}. f(\overline{x})$	Quantifiers
	$\mid$ `emp` $\mid \mathsf{acc}(\overline{\tau}) \mid id(\overline{\tau}) \mid f_1 \& * \& f_2$	Separation logic op
	$\mid f \,?\, f_1 \,:\, f_1$	If-then-else formula

Figure 2.2: Subset of the GRASShopper IR for terms and formulas

and formulas are defined in Figure 2.2 and definitions for the commands are presented in Figure 2.3.

In Figure 2.2, identifiers are defined in a separate domain so we can define uninterpreted functions and symbolic values. Terms defines variables denoted as x and both uninterpreted and interpreted functions are defined by $\mathsf{s}(\overline{\tau})$ which represents an operator denoted by a Symbol applied to an *n-ary* list terms, τ. Symbols have associated sorts with arities, and there are basic sorts such as Bool, Byte, Int, but the syntax of these sorts is omitted from Figure 2.2 for the sake of brevity. Uniary sorts for locations are denoted as $\mathsf{Loc}\langle T \rangle$ where T can be a sort such as $\mathsf{Loc}\langle Int \rangle$. There are Sets and Maps over sorts, and a special Adt sort that is leveraged to represent snapshot trees for representing symbolic values for heap dependent variables. GRASShopper terms disambiguate uninterpreted function and interpreted functions by using an arbitrary identifier $id \in \mathsf{Ident}$ for the former. A few examples of the syntactic forms should illustrate the use of a symbol s to denote both uninterpreted and interpreted functions.

$C \in$ Cmd			Commands
c ::=		$\mathtt{x} := \tau$	Assignment
	|	$C_1; \ldots; C_n$	Sequence control
	|	$f(\tau)$	Procedure call
	|	$C_1 \square \ldots \square C_n$	Non-deterministic Choice
	|	`havoc x`	Havoc
	|	$\mathtt{x} := \mathtt{new}\ \mathtt{T}(\tau)$	Allocation
	|	`free x`	Deallocation
	|	`return x`	Return
	|	`assert f`	Assert
	|	`pure assert f`	Pure assert
	|	`unfold` $p(\tau)$	Predicate unfold
	|	`fold` $p(\tau)$	Predicate fold

Figure 2.3: Subset of the GRASShopper IR for loop-free commands

- Unary minus operator $-$.
- Binary arithmetic operators $+$, $-$, $/$, etc.
- Interpreted functions for manipulation of heap locations are denoted as `read(x.f)` for field reads `x.f` writes `write(f, x, 10)` for a field write `x.f := 10`, for example.
- Uninterpreted functions can be denoted as `fst(s)` to represent the first of a pair s. The interpretation of such a function would have to be described as an axiom and discharged into to the underlying SMT solver.

The GRASShopper syntax for loop-free commands presented in 2.3 is a set of loop-free commands. For the sake of brevity, types are omitted from the presentation even though GRASShopper is statically typed. The Cmd sub-language exposes heap-dependent function calls, predicate applications and method calls using the same syntactic variant $f(\tau)$ but are differentiated by auxiliary metadata about the return arguments as an implementation detail. Where the specific context matters in this discussion, it will be explicitly mentioned if we are dealing with a predicate, procedure, or heap-dependent function call by the surrounding context. Commands such as

a sequence $C_1; \ldots; C_n$ are the classical sequential control, and $C_1 \square \ldots \square C_n$ is a non-deterministic choice. Two unfamiliar commands are likely to be `assert` f and `pure assert` f which both assert that a given formula f holds under a set of constraints. The difference being that assert will determine if a given formula holds under the current permissions and constraints, and `pure assert` only checks the validity of the formula f under the current set of constraints. Note that aside from a semantic difference, there's a syntactic difference where `assert` accepts a formula that can contain SL operators whereas `pure assert` only accepts non-separation logic formulas (pure formulas). GRASShopper also supports parallel assignments such as $\mathtt{x}_1, \ldots, \mathtt{x}_n := \tau_2, \ldots, \tau_n$ for the assignment command and for procedure calls and return commands. We omit this notation from the syntax for loop-free commands to simplify the presentation, but any rule that would involve an assignment, `havoc`, return, etc does support this multi-arity behavior.

2.2 Language extensions

The work done under this thesis augmented the GRASShopper IR in order to provide more powerful features that are also supported in Viper [18]. The language changes to GRASShopper were the following

- Addition of if-then-else expressions, $f \ ? \ \tau_1 \ : \ \tau_2$. Where the branch condition is a pure formula $f \in \mathsf{Form}$.
- Addition of `unfold` and `fold` commands to unroll a predicate `unfold` $length(\mathtt{x.next}, \mathtt{y})$ or roll up a predicate.
- Modified access predicates to cover individual fields of structures rather than assuming access to all fields defined in a given structure. Before this change GRASShopper would grant access to a location type as `acc(x)`, and after this change finer-grained access can be specified as `acc(x.f)`. Note that Viper uses fractional permissions to reason about concurrency,

and thus can grant partial access to a field as acc(x.f, 1/2), but our adaptation of this idea assumes whole permissions on a given field.

- Addition of `unfolding` expression that will unroll a predicate instance and continue the symbolic execution in the state carrying the emitted predicate instance and then remove the predicate's formula after the `unfolding` expression terminates.

2.3 Notation

We begin with some basic definitions and common notation used in the formalization of the symbolic execution rules. We use the standard notation for sets where inclusion of a member x in a set S is $x \in S$, the empty set is denoted by $\varnothing$ and set union and intersection are $\cup$ and $\cap$, respectively. Set literals are denoted by $\{a, b, c\}$. When a multiset can be implied by its surrounding context we use the same notations as sets. We introduce a power set using the notation 2^S as the set of all subsets of S. Explicit notation for multisets will use $\{x_1 \rightarrowtail i_1, \ldots, x_n \rightarrowtail i_n\}$ for the multiset containing i_1 occurrences of x_1 and i_2 occurrences of x_2, etc. For a multiset S, we write $S(x)$ to denote the number of occurrences of $x \in S$.

The symbolic state will track a mapping from program variables to symbolic values. This mapping is naturally a map so we will need definitions of insertion $m[k \mapsto v]$ which creates a new map m' where k maps to a value v. In a context where k already exists a map update is denoted by $m(k) = v$.

A total function from A to B is written as $f : A \rightarrow B$ and $f : A \nrightarrow B$ for a partial function from A to B.

We define a (finite) sequence $\xi \in S^*$ of elements of a set S which is defined recursively with the syntax $\xi ::= \epsilon \mid \xi \cdot v$ with $v_1, v_2, \ldots, v_n$ when $n \geq 0$ and is the empty sequence $\epsilon \in S^*$ when $n = 0$. We also define the concatenation $\xi \cdot s$ of a sequence ξ with an element $s \in S$ (such that $\epsilon \cdot s = s$ is a single-element sequence).

Iterated operators are used with sets and sequences, for example, $\bigwedge \xi$ denotes the conjunction of all elements of the set or sequence ξ. If an iterated operator is used in conjunction with other operators such as an implication $\bigwedge \omega s \implies \phi$, the iterated operator is left-associative, namely $(\bigwedge \omega s \implies \phi) \Leftrightarrow ((\omega s_1 \wedge \ldots \wedge \omega s_n) \implies \phi))$.

During the computation of a symbolic execution, an identifier or symbol is `fresh` if it differs syntactically from any other identifier or symbol in use. When a fresh identifier or symbol is needed `fresh` is used. It can also be used in places where any type of identifier is possible. In this case $b := \texttt{fresh}$ is used.

We let $\mathcal{R}$ be a type for symbolic verification results with the same meaning as the verification result in Viper [18]. A verification result $\mathcal{R}$ takes on two values, `success` or `failure`, and can be composed by the iterated conjunction $\bigwedge$. For example, if a step during the SE has a return type of $\mathcal{R}$, then `success` $\bigwedge \mathcal{R}$ is $\mathcal{R}$ and `failure` $\bigwedge \mathcal{R}$ short-circuits and will evaluate to `failure` no matter what value $\mathcal{R}$ holds.

2.4 Verification as an SMT Problem

One way to view the verification problem is to assign a logical meaning to the program using Hoare triples [11]. A Hoare triple is a formula $\{P\}Q\{R\}$, where P is a precondition Q is a program and R is a post-condition. The correctness of such a program is checked by an SMT solver by reducing the program to a set of logical formulas using a technique such as VCG (or SE) to verify that the pre-condition P and the program constraints in Q, entail the post-condition R. Suppose the pre-condition formula and the constraints implied by the program Q reduce to a formula S, then the correctness of the program will be checked by querying an SMT solver to determine if the formula $S \wedge \neg R$ is satisfiable. If the solver reports that the formula is *unsatisfiable*, then we know that the program is correct. Otherwise, we know that R doesn't hold. GRASShopper supports queries against the CVC4 [2] and Z3 [6] SMT solvers solvers. All of the development

and examples were checked using the Z3 SMT solver.

3 Design & Theory

We present the design and theory behind the implementation of the SE engine for GRASShopper which was largely inspired by Malte Schwerhoff's PhD thesis [24], which describes Viper's SE engine. The specific contributions made by this work presented under this chapter are the mathematical presentation of the operational semantics and semantic domains in Sections 3.2 and 3.1, the snapshot encoding extension to support structures that have fields that can carry any GRASShopper type in Section 3.3, and the discussion around the interplay between function axiom generation and the join rule for evaluating `unfolding` expressions presented in Section 3.8. All other sections around the description of the SE rules for `exec`, `produce`, `consume`, and `eval` rules mirror the Viper presentation but may differ slightly due to variations between the semantics of Viper and GRASShopper.

3.1 Semantic Domains

In order to define the execution rules for the symbolic interpreter, we need to define the set of domains that compose the symbolic state (see Figure 3.1) and define the operations to manipulate the domains during the symbolic execution.

A key idea in symbolic execution is that concrete values of program variables are replaced by symbolic values. Symbolic values are simply Terms that don't refer to any program variable or a heap location. For heap locations, a sub-type of Terms called *snapshots* are used to denote the

v	$\in$ SymbVal	Symbolic values
s	$\in$ Snap	Snapshot
$\varphi = \langle id, v, \xi \rangle$	$\in$ PCScope $\triangleq$ *scope*(Ident $\times$ SymbVal $\times$ Stack)	Path condition scope
ξ	$\in$ Stack $\triangleq$ (SymbVal*)	Stacks
π	$\in$ PCStack $\triangleq$ (PCScope*)	Path condition stack
$\kappa = fld\langle id, r, s\rangle \cup$ $pred\langle id, \overline{v}, s\rangle$	$\in$ HeapChunk $\triangleq$ *fld*(Ident $\times$ Ident $\times$ Snap) $\cup$ *pred*(Ident $\times$ SymbVal* $\times$ Snap)	Heap chunk
h	$\in$ Heap $\triangleq$ {HeapChunk}	Symbolic heap
$\sigma = \langle \gamma, \pi, h \rangle$	$\in$ SymbState $\triangleq$ SymbStore $\times$ PCStack $\times$ Heap	Symbolic state
γ	$\in$ SymbStore $\triangleq$ Terms $\rightarrow\!\!\!\!\!\!\rightarrow$ SymbVal	Symbolic store

Figure 3.1: Semantic domains of the GRASShopper symbolic execution semantics

symbolic values of heap locations. Thus, we let $v \in$ SymbVal be a sub-type of the Terms domain defined in Figure 2.2. We refer to the elements of SymbVal as *symbolic values*. Symbolic values are differentiated from program variables by their font in this text. Namely, x is a program variable and x is a symbolic value. We also define $s \in$ Snap to be a sub-type of SymbVals to represent both symbolic values for struct fields and heap-dependent function footprints. These details around snapshots are defined in more detail in Section 3.3.

To track the mappings from program variables to symbolic values we define a *symbolic store*, $\gamma \in$ SymbStore $\triangleq$ Terms $\rightarrow\!\!\!\!\!\!\rightarrow$ SymbVal.

We let $\varphi \in$ PCScope $\triangleq \langle id, v, \xi \rangle$ be defined as a *path condition scope*. A path condition scope is a 3-tuple that carries a unique, fresh identifier for a scope, a *branch condition*, and a stack $\xi \in$ Stack $\triangleq$ (SymbVal*) consisting of a sequence of symbolic values v. These PCScopes represent the *path conditions* collected under a given *branch condition*.

A *path condition* stack records the set of constraints collected during the symbolic execution under *branch conditions* and is $\pi \in$ PCStack $\triangleq$ (PCScope*), a sequence of 0 or more PCScopes.

To support permission-based reasoning over field references x.f and predicates we define a heap chunk $\kappa = fld\langle id, r, s\rangle \cup pred\langle id, \overline{v}, s\rangle \in$ HeapChunk $\triangleq$ *fld*(Ident$\times$Ident$\times$Snap)$\cup$*pred*(Ident$\times$ SymbVal* $\times$ Snap). A heap chunk $\kappa =$

$mathit fld\langle id, r, s\rangle$ encodes that permissions are given to a field id with receiver r and has a corresponding symbolic value v to represent the value located at $r.id$. A heap chunk $\kappa = pred\langle id, \overline{v}, w\rangle$ encodes permissions to a predicate instance named id with arguments that have been replaced by symbolic values $\overline{v}$ and the footprint of the predicate instance is the snapshot (or synonymously a symbolic value) w. A symbolic heap is a multiset of heap chunks, $h \in \mathsf{Heap} \triangleq \{\mathsf{HeapChunk}\}$.

The symbolic state is defined as $\sigma = \langle \gamma, \pi, h\rangle \in \mathsf{SymbState} \triangleq \mathsf{SymbStore} \times \mathsf{PCStack} \times \mathsf{Heap}$. Each step taken in the symbolic interpreter's execution will produce a fresh state that will be used to discharge queries to the SMT solver at specific points to verify the program.

3.2 Operational Semantics

We present the operational semantics for the symbolic execution of GRASShopper. The operational semantics define the transitions the symbolic interpreter will take in order to verify methods and heap-dependent functions independently. The rules are presented using continuation passing style instead of the usual inference rule style because the definitions clearly state what the remainder of the computation is. For more detail on continuation passing style we refer the reader to [1]. The rules can be represented by the following primitives

$$\begin{aligned}
&\mathtt{exec} : \mathsf{SymbState} \to C \to (\mathsf{SymbState} \to \mathcal{R}) \to \mathcal{R}\\
&\mathtt{produce} : \mathsf{SymbState} \to \mathsf{Forms} \to \mathsf{Snap} \to (\mathsf{SymbState} \to \mathcal{R}) \to \mathcal{R}\\
&\mathtt{consume} : \mathsf{SymbState} \to \mathsf{Heap} \to \mathsf{Forms} \to (\mathsf{SymbState} \to \mathsf{Heap} \to \mathsf{Snap} \to \mathcal{R}) \to \mathcal{R}\\
&\mathtt{eval} : \mathsf{SymbState} \to \mathsf{Terms} \to (\mathsf{SymbState} \to \mathsf{Terms} \to \mathcal{R}) \to \mathcal{R}
\end{aligned} \tag{3.1}$$

The exec rule symbolically executes a command $c \in C$ in the current symbolic state and the continuation $Q \triangleq (\mathsf{SymbState} \to \mathcal{R})$ will execute the remainder in a new state and return a verification result $\mathcal{R}$ and the overall result of exec will be returned after the continuation termi-

nates. There are two dual rules that are used to exhale and inhale permissions while executing symbolic Terms, `produce` and `consume`. The `produce` rules will emit permissions in the current symbolic state corresponding to the second argument assertion Forms and the third Snap argument. The continuation for `produce` will execute the remainder of the computation in a new state and return a verification result. The `consume` rules will remove permissions from the Heap and take the corresponding snapshot $s \in$ Snap for the HeapChunk that was removed and pass the snapshot located in the third argument of, $k = fld\langle id, r, s\rangle$ and $pred\langle id, \overline{v}, s\rangle$ to its continuation $Q \triangleq$ (SymbState $\rightarrow$ Heap $\rightarrow$ Snap $\rightarrow \mathcal{R}$). The second parameter of sort Heap of `consume` is a bit peculiar since the symbolic state carries a heap. This second heap is a copy of the heap in the symbolic state to prevent terms such as `acc(x.f) &*& write(x, f)` from failing. Symbolic Terms are evaluated using the `eval` primitive which evaluates a Term in the current SymbState and passes the evaluated Term to the continuation.

3.2.1 Execution Rules

We present a subset of the execution rules, denoted as `exec` in Equation 3.1, rules for the GRASShopper Cmds domain. The first rule defined in Figure 3.2 for the assignment `x` $:= \tau$ will illustrate how program variables are mapped to symbolic values which can then be later used in more complex rules that may need to build constraints over symbolic values. The assignment rule evaluates τ to a symbolic value τ' and passes it along with the a possibly updated state σ_2 to the continuation of `eval`. The remainder of the evaluation will update σ_2's symbolic store γ such that `x` now maps to τ' and passes this state to the continuation Q.

The next rule defined in Figure 3.2 on line 5 illustrates how the symbolic execution implements permission-based checking of heap-dependent state. The field write will first evaluate τ to a symbolic value τ' and then scan the heap to check if sufficient permissions on `write(x, f)` exist. This is implemented by *consuming* the access predicate Term, `acc(x.f)`. If there's a field chunk in the symbolic heap, then the execution will continue and *produce* the access predicate on the heap re-

```
1  exec(σ, x := τ, Q) =
2    eval(σ, τ, (λ σ2, τ'.
3      Q(σ2{γ := σ2.γ[x ↦ τ']})
4
5  exec(σ, write(x, f, τ), Q) =
6    eval(σ, τ, (λ σ1 τ'.
7      consume(σ1, σ1.h, acc(x.f), (λ σ2 h' s.
8        produce(σ2, acc(x.f) &*& read(x, f) == τ', fresh, Q)))))
9
10 exec(σ, x := new T(τ), Q) =
11   exec(σ, havoc x, (λ σ2.
12     produce(σ2, acc(x, τ), fresh, Q)))
13
14 exec(σ, havoc x, Q) =
15   Q(σ{γ := σ.γ[x ↦ fresh]})
16
17 exec(σ, x := f(τ), Q) =
18   eval(σ, τ, (λ σ2τ'.
19     consume(σ2, σ.h, f_pre[τ'/τ], (λ σ3, h2, s.
20       exec(σ3, havoc x, (λ σ4.
21         produce(σ4, f_post[τ'/τ][y/x], fresh, (λ σ5. Q(σ5)))))))))
22
23 exec(σ, fold p(τ), Q) =
24   eval(σ, τ, (λ σ2τ'.
25     consume(σ2, σ.h, p_body[τ'/τ], (λ σ3, _, s.
26       produce(σ3, p'(τ'), s, (λ σ4. Q(σ4)))))))
27
28 exec(σ, unfold p(τ), Q) =
29   eval(σ, τ, (λ σ2τ'.
30     consume(σ2, σ.h, p'(τ'), (λ σ3, _, s.
31       produce(σ3, p_body[τ'/τ], s, (λ σ4. Q(σ4)))))))
```

Figure 3.2: Execution rules for the GRASShopper Cmd domain.

gion, and subsequently push a constraint back on to the stack π, `acc(x.f) &*& read(x, f) == τ'` with a fresh, empty snapshot.

The rule for memory allocation in Figure 3.2 on line 10 is straightforward and since the allocation command is used in conjunction with a variable assignment the first step is to havoc `x` and then push an access predicate onto the heap.

Method calls $f(\tau)$ are symbolically executed by evaluating the formal parameters and then substituting the actual symbolic formal parameters τ' in place of the formal parameters in the function's precondition. After the substitution, the substituted f_{pre} is consumed to establish the

precondition. The rule then havocs the return value x and then repeats the substitution for the formal arguments and substitutes the actual return value x for the formal return parameters y.

The fold and unfold rules in Figure 3.2 on lines 23 and 28, respectively, produce or consume the symbolic predicate chunk, which is denoted as a primed application $p'(\tau')$ to stress the symbolic nature, relative to consuming or producing the body formula. This rule ensures that the formula covered by the predicate is available in the symbolic state for the continuation.

3.2.2 Produce & Consume Rules

Produce rules are designed to push state into the path-condition stack π or the symbolic heap h. We present a subset of the rules to illustrate how the symbolic state is manipulated. The most basic produce rule is producing a pure formula such as x == 10. Figure 3.3 line 1 shows that producing a pure formula pure(f) evaluates f to f' and then passes it to the continuation. The continuation will then push f' onto π and an additional constraint $f' == unit$. This signifies that the formula is pure and doesn't correspond to a heap-dependent snapshot.

```
1  produce(σ, pure(f), unit, Q) =
2    eval(σ, f, (λ σ2, f'.
3      Q(σ2{π := σ2.π · f' · (f' == unit)})))
4
5  produce(σ, acc(x.f), s, Q) =
6    eval(σ, x, (λ σ2, x'.
7      Q(σ2{h := σ2.h · fld⟨f, x', s⟩})))
8
9  produce(σ, f1 &*& f2, s, Q) =
10   produce(σ, f1, fst(s) (λ σ'.
11     produce(σ, f2, snd(s), Q)))
12
13 produce(σ, f : f1 ? f2, s, Q) =
14   eval(σ, f, (λ σ2, f'.
15     branch(σ2, f'
16         (λ σ3, f'. produce(σ3, f1, s, Q))
17         (λ σ3, f'. produce(σ3, f2, s, Q)))))
```

Figure 3.3: A subset of produce rules for the GRASShopper Formula domain.

Producing an access predicate for a field read acc(x.f) in Figure 3.3 line 5 evaluates the re-

ceiver x and then pushes a field chunk $fld\langle x', f, s\rangle$ onto the symbolic heap h. The recursive structure of GRASShopper separation logic formulas is handled by the produce rule in line 9. Producing an if-then-else expression introduces a branching helper that will only explore feasible branches if the branching condition formula f is satisfiable by the current symbolic state. We will discuss this rule in more detail in Section 3.8.

We now present rules for consuming or removing permissions from the symbolic state. The consume rules are the dual of produce rules and we present analogous rules to each produce rule presented in Figure 3.3.

```
1  consume(σ, h, s, pure(f), Q) =
2    eval(σ, f, (λ σ2, h', f'.
3     if pc-all(σ2.π) ⊢ f' then
4       Q(σ2, h, unit)
5     else
6       failure))
7
8  consume(σ, h, acc(x.f), s, Q) =
9    eval(σ, x, (λ σ2, x'.
10       if x ∈ dom(σ2.h) then
11         let fld⟨_, _, s⟩, h' = heap-remove(x, id, h) in
12           Q(σ2, h', s)
13       else
14         failure
15
16 consume(σ1, h1, f1 &*& f2, Q) =
17   consume(σ1, h1, f1, (λ σ2, h2, s1.
18     consume(σ2, h2, f2, (λ σ3, h3, s2.
19       Q(σ3, h3, pair(s1, s2))))))
20
21 consume(σ, h, f : f1 ? f2, Q) =
22   eval(σ, f, (λ σ2, f'.
23     branch(σ2, f'
24       (λ σ3, f'. consume(σ3, f1, Q))
25       (λ σ3, f'. consume(σ3, f2, Q)))))
```

Figure 3.4: A subset of the consume rules for the GRASShopper Formula domain.

Consuming a pure formula pure(f), is presented in Figure 3.4. This rule on line 1 first evaluates f to f' and then passes it to the continuation of eval. The continuation will then flatten all of the formulas in $\sigma_2.\pi$ and then check if the collected path conditions entail f'. If that entailment

holds, then the continuation is called with the *unit* snapshot since consume rules pass snapshots to continuations.

Consuming an access predicate for a field read `acc(x.f)` in Figure 3.4 line 8 evaluates the receiver `x` and checks if the receiver is in the $dom(h)$ if so then the field chunk is removed from the heap and the matching field chunk's snapshot is then passed to the continuation. Otherwise, if the field chunk isn't present in the heap, it means that sufficient permissions for that region aren't allowed and the verification should fail.

Consuming the recursive structure of GRASShopper separation logic formulas is handled by the consume rule in Figure 3.4 on line 16, and is the same as the corresponding produce rule except that we consume the left-hand side of the separation star and the right-hand side formula. Consuming an if-then-else expression is analogous to the produce rule that introduces the branching helper in Figure 3.3 line 13.

3.2.3 Symbolic Expression Evaluation Rules

Here we present a subset of expression evaluation rules for the GRASShopper Term and Form domains. These rules closely follow the Viper expression evaluation rules so we will briefly discuss the subset of rules presented in Figure 3.5. The first rule in line 1 describes a variable is evaluated. To evaluate a variable one simply looks up the variable in the current symbolic state's symbolic store, γ, and passes it to the continuation. A variation on this rule can lazily havoc a variable `x` if one doesn't exist in $\sigma.\gamma$.

To evaluate a binary operator in line 5 we recursively evaluate each sub-expression and then stitch the symbolic binary operator, `bop'` (f_1', f_2') back together and pass it to the continuation. Recall that from the syntax of GRASShopper in Section 2.1 `bop` abstracts the usual binary boolean and numeric infix operators.

The evaluation rule to evaluate a field read, `read(x, f)`, in line 10 of Figure 3.5 will evaluate a read on a field `f` with receiver `x`. The rule checks if the receiver is in $dom(\sigma.h)$ and if so it will

```
1  eval(σ, x, Q) =
2    let v = σ.γ(x) in
3      Q(σ, v)
4
5  eval(σ, bop(f1, f2), Q) =
6    eval(σ, f1, (λ σ1, f1'.
7      eval(σ1, f2, (λ σ2, f2'.
8        Q(σ2, bop'(f1', f2'))))))
9
10 eval(σ, read(x, f), Q) =
11   eval(σ, x, (λ σ2, x'.
12       if x ∈ dom(σ.h) then
13         let fld⟨_, _, s⟩ = σ.h(x) in
14           Q(σ2, s)
15       else
16         failure
17
18 eval(σ1, f1 &*& f2, Q) =
19   eval(σ1, f1, (λ σ2, f1'.
20     eval(σ2, f2, (λ σ3, f2'.
21       Q(σ3, f1' * f2')
22
23 eval(σ, old(τ), Q) =
24   eval(σ{h := h_old(σ)}, τ, (λ σ2, τ'.
25     Q(σ2{h := σ.h}, τ')))
```

Figure 3.5: A subset of the symbolic expression rules for the GRASShopper Terms domain.

bind the snapshot, s carried in the corresponding field chunk in the heap of the current state, σ. Otherwise if a field chunk isn't found then an access permission on the field wasn't produced earlier and the symbolic execution will fail.

The next rule in line 18 is straightforward and it will recursively evaluate the left and right sub-trees of a separation conjunction of two formulas, f_1 `&*&` f_2. After the evaluation is complete the symbolic formula for each sub-tree is passed to their respective continuations and the symbolic separation conjunction is passed to Q.

Lastly, the rule to evaluate an `old` expression in line 23 will evaluate the term τ by re-instating the old heap, h_{old} which is left out of the discussion for the sake of brevity, as the heap h in the current state, and then re-instate the current heap h when calling the continuation. The symbolic execution engine for GRASShopper also supports evaluation of ternary if-then-else expressions,

Viper's `unfolding` expression which are translated verbatim from Viper, thus we refer the reader to [24].

3.3 Snapshot Encoding

In order to support modular verification of predicates and procedures, we need a way to independently check that a procedure only depends on the heap locations covered by it's precondition. This idea was originally solved by Samans et al. [26] by implementing *framing* using snapshots. To illustrate the idea behind snapshots and to provide background into how snapshots are encoded for the GRASShopper symbolic execution, consider a GRASShopper program that implements a simple counter in Figure 3.6.

In order to verify that a call to `incr` which uses facts about the heap encoded in `counter` guarantees that the only memory locations that are mutated are the counter's `count` member, we need a way to reason that about the structure of the memory locations that are covered by `counter`. This is the basic idea behind framing. Samans [26] first introduced the idea of supporting framing reasoning using snapshots for predicates using predicate chunk snapshots. Figure 3.7 annotates each program point with the symbolic state.

We can see in line 5 that we have permissions on the counter predicate which is denoted by the presence of the predicate chunk $pred\langle counter, [v_1, v_2], snap1\rangle$, in the symbolic heap h. The predicate chunk carries $snap_1 \in Snap$ and is the snapshot corresponding to the memory location covered by the `counter` predicate. As the execution continues, and we `unfold` the counter predicate (this is formally defined in Section 3.2) this will scan h for a predicate chunk that matches the `counter(c, v)` predicate instance, and it will replace it with the separation logic formula in the body of `counter`. Notice in line 7 that after the `counter` predicate is unfolded we have a constraint on the top scope of π, $v_2 == fSnap_{Int}(fst(snap_1))$. This equality constraint encodes the right-hand side of the separation logic formula `acc(c.count) &*& v == c.count`. The field chunk

```
1  struct Counter {
2    var count: Int;
3  }
4  predicate counter(c: Counter, v: Int) {
5    acc(c.count) &*& v == c.count
6  }
7
8  procedure incr(c: Counter, ghost v: Int)
9    requires counter(c, v)
10   ensures counter(c, v+1)
11 {
12   unfold counter(c, v);
13   c.count := c.count + 1;
14   fold counter(c, v+1);
15 }
16
17 procedure create() returns (c: Counter)
18   requires emp
19   ensures counter(c, 0)
20 {
21   c := new Counter();
22   c.count := 0;
23   fold counter(c, 0);
24 }
```

Figure 3.6: A counter predicate definition and a procedure that increments the counter by one The ghost variable v is solely used to verify that the counter is incremented correctly. The predicate counter gives permissions to access the c.count member of the Counter and it enforces that the counter's count is equal to v. The counter predicate can be used in the precondition of incr and folded and unfolded in the body.

inherently encodes the mapping between c.count and the snapshot $fst(snap_1)$. In order for the constraint to be an equality over integers since we know from the definition of counter in Figure 3.6 that the type of v is Int and we know that $fst(snap_1) \in$ Snap. Therefore, we need a way to encode that $fst(snap_1)$ is a sub-type of Int which is encoded in the function $fSnap_{Int}$.

Now to illustrate framing we consider a client program in Figure 3.8 that allocates two counters, c1 and c2, and increments c1 by 1 and then asserts that two separate counter regions are disjoint. When this is the case we can conclude memory safety of c2 because only c1 has been mutated.

We show a point-in-time instance of the symbolic state after both counters are allocated and

```
1  procedure incr(c: Counter, ghost v: Int)
2    requires counter(c, v)
3    ensures counter(c, v+1)
4  {
5  {σ{γ = {c ↦ v1, v ↦ v2}; h = [pred⟨counter, [v1, v2], snap1⟩]; π = []}}
6    unfold counter(c, v);
7    { σ{γ = {c ↦ v1, v ↦ v2};
       h = [fld⟨count, v1, fst(snap1)⟩];
       π = [(scope1, true, [snd(snap1) == emp; v2 == fSnapInt(fst(snap1))]}
8    c.count := c.count + 1;
9    { σ{γ = {c ↦ v1, v ↦ v2};
       h = [fld⟨count, v1, fst(snap1)⟩];
       π = [(scope1, true, [fSnapInt(fst(snap1)) == fSnapInt(fst(snap1) + 1;
                            snd(snap1) == emp; v2 == fSnapInt(fst(snap1))]}
10   fold counter(c, v+1);
11   { σ{γ = {c ↦ v1, v ↦ v2};
       h = [pred⟨counter, [v1, v2], snap1⟩];
       π = [(scope1, true, [fSnapInt(fst(snap1)) == fSnapInt(fst(snap1) + 1;
                            snd(snap1) == emp; v2 == fSnapInt(fst(snap1))]}
12 }
```

Figure 3.7: Symbolic state during verification of `incr`.

c1 has been incremented by 1. The symbolic state in Figure 3.8 line 10 shows there each predicate chunk in the heap have unique snapshots s_1, s_2, and s_3 that represent the footprint of each counter. The assertion on line 11 will result in an SMT query that is shown in line 14 to check if $s_3 \neq s_2$ which fails because each snapshot is unique and thus when each counter predicate is unfolded there will be different snapshot values used in the subsequent reasoning, hence the counters are properly framed.

3.3.1 Injective Axioms

One of the most interesting technical contributions of this work is the use of injective functions to map snapshot types to program types of the variables they represent. This is necessary because Viper has a single `Ref` type to represent a reference where as GRASShopper supports pointers to any type. An example where this occurs is when a struct field can take an arbitrary type and we

```
1  procedure client()
2    requires emp
3    ensures emp
4  {
5    var c1 := create();
6    var c2 := create();
7
8    incr(c1, 1);
9
10   { σ{γ = {c1 ↦ v1, c2 ↦ v2};
     { h = [pred⟨counter, [v1, 1], s3⟩, pred⟨counter, [v2, 0], s2⟩, pred⟨counter, [v1, 0], s1⟩];
     { π = []}
11   assert counter(c1, 1) &*& counter(c2, 0);
12
13   // SMT query
14   {∀s1, s2, s3 ∈ Snap. s3 ≠ s2}
15
16   delete(c1, 1);
17   delete(c2, 0);
18 }
```

Figure 3.8: A counter client program with annotations of the symbolic state to to demonstrate framing.

need a symbolic value to represent that heap location with a given type to execute a field read or write. The approach taken in this work is to use injective functions to encode the mapping from the GRASShopper type to the Snap type and vice versa.

Recall, that a function $f \in S_1 \leftarrow S_2$ is injective iff $\forall x_1 \in S_1 \,.\, \forall x_2 \in S_2 \,.\, x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$. Each symbolic value that corresponds to a location on the heap will emit an axiom using an injective function of type $f_{inj} : \text{Snap} \rightarrow T$ where T is the type of the program variable,

$$\forall x \in \text{Snap} \,.\, f_{inj}^{-1}(f_{inj}(x)) = x$$

For example, the axiom that will be emitted to the SMT solver for the snapshots corresponding to the `c.counter` locations in Figure 3.7 will be

$$\forall x \in \text{Snap} \,.\, {fSnap_{Int}}^{-1}(fSnap_{Int}(x)) = x$$

where, $fSnap_{Int}$ is an uninterpreted symbol with a sort Int.

3.3.2 Snapshots as ADTs

Viper encodes snapshots as a sub-type of symbolic values and encodes a signature of snapshot-related functions:

$$\begin{aligned} unit &: \mathsf{Snap} \\ pair &: \mathsf{Snap} \rightarrow \mathsf{Snap} \rightarrow \mathsf{Snap} \\ fst &: \mathsf{Snap} \rightarrow \mathsf{Snap} \\ snd &: \mathsf{Snap} \rightarrow \mathsf{Snap} \end{aligned}$$

where *unit* is the empty snapshot, and *fst* and *snd* are interpreted as,

$$\forall s_1, s_2 \in \mathsf{Snap} \;.\; fst(pair(s_1, s_2)) = s_1 \wedge snd(pair(s_1, s_2)) = s_2$$

A handy notation for multiple applications of *fst* and *snd* operations such as $fst(fst(s_1))$ or $fst(fst(fst(snd(snd(s_1)))))$ is to adapt a power function notation to mean the repeated application where $f^0(s) \triangleq s$ and $f^{n+1}(s) \triangleq f(f^n(s))$. By applying this definition we can write $fst(fst(s_1))$ as $fst^2(s_1)$ and $fst(fst(fst(snd(snd(s_1)))))$ as $fst^3(snd^2(s_1))$.

We take a different approach by encoding snapshots as ADTs into the SMT solver since GRASShopper already had an ADT type to encode binary trees for the VCG backend. The encoding is straightforward,

$$\texttt{datatype}\ \mathsf{Snap} = \texttt{emp} \mid \texttt{tree}(fst : \mathsf{Snap}, snd : \mathsf{Snap})$$

3.4 State Consolidation

A known problem with SE engines that maintain separate state for pure formulas via a stack and heap-based permissions in a symbolic heap are possible cases when aliasing can cause heap incompleteness. Both Smallfoot-style verifiers and Viper encounter this issue, see [3, 24]. This kind of incompleteness is different from Rice's undecidability and is more related to the state not being able to encode aliasing in these disjoint data structures.

An example of where incompleteness arises is when variables are aliased. Consider the following program where aliasing occurs in Figure 3.9,

```
1  struct Node {
2    var data: Int;
3    var next: Node;
4  }
5
6  procedure alias1(x: Node) returns (y: Node)
7      requires acc(x.next)
8      ensures acc(x.next) &*& acc(y.next)
9  {
10     y := new Node();
11     x := y;
12     // Passes
13     pure assert x == y;
14 }
```

Figure 3.9: An example of incompleteness due to aliasing that can be resolved by state consolidation.

State consolidation can overcome incompleteness in `alias1` because at the point where the formula is checked against the symbolic state in line 13 the PCStack is empty so the SMT solver can never know that `x == y`. The state consolidation algorithm will emit the constraint `x == y` to the symbolic state and the assertion will pass.

However, this state consolidation approach isn't a silver bullet as the verification of `alias2` in Figure 3.10 fails due to there being a disjunctive alias `z == y || z == x` regardless. Viper discusses that this type of disjunctive aliasing can only be overcome by using ghost code to force

branching, so this is an expected failure.

```
1 procedure alias2(x: Node, y: Node) returns (z: Node)
2     requires acc(x.data) &*& acc(y.data)
3     ensures acc(x.data) &*& acc(y.data) &*& acc(z.data)
4 {
5     // Fails
6     assert (z == y || z == x);
7 }
```

Figure 3.10: An example of incompleteness due to aliasing that cannot be resolved by state consolidation.

Since GRASShopper doesn't support fractional permissions like Viper's `acc(x.f, 1/2)` the state consolidation operation in GRASShopper is simplified. We only implemented a function to infer disequalities in Figure 3.11 and we add an additional constraint to ensure each receiver of a field chunk, $fld\langle x, \_, \_\rangle$, is non-null.

```
1 infer-disequalities(h, π) =
2 let s = {(x_i, y_i) | fld⟨x_i, f_i', s_i⟩ ∈ h ∧ fld⟨y_i, f_i, s_i'⟩ ∈ h ∧ f_i = f_i'} in
3   fold_l((λπ', (q_i, r_i). pc-add(π', q_i ≠ r_i ∧ x_i ≠ null)), π, s)
```

Figure 3.11: A function to infer disequalities from aliased heap chunks.

The approach taken in this work around where state consolidation occurs is done optimistically at every SMT call. A more sophisticated approach would be to only do state consolidation when an SMT query fails while searching for matching heap chunks.

3.5 Well-Formedness and Validity Checks

The main driver functions that begin checking each procedure are implemented as a set of functions that first ensure that the specifications are well-formed. Well-formedness is essential to ensure that the verification of code contained in the body of a procedure, function, or predicate doesn't fail due to a buggy specification. For instance if one is to specify that the value of the next node in a linked list should be 10 a correct specification would be

$$\texttt{acc(x.next) \&*\& x.next.data == 10} \tag{3.2}$$

In Equation 3.2, the formula must specify that access is granted to `x.next` before we can access the data member and push a constraint onto the stack. An incorrect specification could be,

$$\texttt{x.next.data == 10 \&*\& acc(x.next)} \tag{3.3}$$

Equation 3.3 is ill-formed because the evaluation of formulas is a strict left-to-right evaluation, and in order to produce a formula that relies on a heap-dependent field the symbolic value must be available in the symbolic heap. This is because the constraint `x.next.data == 10` must be pushed onto the stack using the symbolic value for `x.next.data` which is carried in the heap chunk $\kappa = \mathit{fld}\langle x, \mathit{next}, s\rangle$.

The set of verification functions that check well-formedness and that each procedure satisfies its specification are a bit different that the well-formedness checks in Viper, but are similar in spirit. The main difference between these functions in GRASShopper and Viper are due to the fact that GRASShopper doesn't support Viper's inhale/exhale variant permissions [24], and are thus simpler. The rules for checking well-formedness of pre- and post-conditions and the validity of procedures, functions and predicates are presented in Figure 3.12. Each `verify` function returns a sequence of formulas corresponding to a set of axioms and an error. The axiom sequence is only populated when verifying heap-dependent functions which also axiomatize each checked function while doing well-formedness checks and validity checks as an optimization. The ultimately generated axiom is returned from the `verify` function so it can be later emitted to the SMT solver. So $\texttt{verify-proc}(\sigma, \overline{\mathit{args}}, \mathit{pre}, \mathit{post}, \overline{\mathit{cmd}}) \in \mathsf{SymbState}\times\mathsf{Terms}\times\mathsf{Form}\times\mathit{Form}\times\mathsf{Cmds} \nrightarrow \mathsf{Form}\times\mathcal{R}$. The function signature for verifying predicates is a bit different compared to `verify-proc` since predicates don't have a specification and the only piece of the predicate that needs to be checked

for well-formedness is the predicate body which is a Term, so `verify-pred`$(\sigma, \overline{args}, body) \in$ SymbState $\times$ Terms $\times$ Form $\twoheadrightarrow$ Form $\times \mathcal{R}$, and since the return value of a heap-dependent function is needed to axiomatize the function; `verify-func`$(\sigma, id, \overline{args}, ret, pre, post, body) \in$ SymbState $\times$ Ident $\times$ Terms $\times$ Term $\times$ Form $\times$ Form $\times$ Form $\twoheadrightarrow$ Form $\times \mathcal{R}$

```
1  verify-proc(σ, args, pre, post, c) =
2  produce(σ, pre, fresh, (λ σ1.
3     produce(σ1, post, fresh, (λ σ2.
4       exec(σ1, c, (λ σ3.
5       consume(σ3, post, (λ σ4. (ϵ, success)))))))))
6
7  verify-pred(σ, args, body) =
8    exec(σ, havoc args, (λ σ1.
9      produce(σ1, body, fresh, (λ σ2.
10       (ϵ, success)))))
11
12
13 verify-func(σ, id, args, ret, pre, post, body) =
14   produce(σ, pre, fresh, (λ σ1.
15     exec(σ1, havoc ret, (λ σ2.
16      produce(σ2, post, fresh (λ σ3.
17        produce(σ3, c, (λ σ4.
18          let f = axiom(σ2, σ3, id, args, ret) in
19          consume(σ4, post, (λ σ5. (ϵ · f, success)))))))))))
```

Figure 3.12: Well-formedness and validity checking rules for procedures, functions, and predicates.

We now explain the reasoning behind the steps in the `verify` rules presented in Figure 3.12. The function `verify-proc` begins by producing the pre- and post-conditions for well-formedness checks in lines 2 and 3. Once the pre- and post-conditions are validated then the body of the procedure is executed in line 4 using the symbolic state after the pre-condition has been produced, σ_1. Lastly, the verification that the precondition and the body entail the post-condition is checked by consuming the post-condition in line 5 under σ_3 which is the state after the body has been executed. If this succeeds, then the result is `success` and the verification passes. Similar reasoning is applied to `verify-pred` except that the only goal is to check well-formedness of the body and then continue in a state that contains the result of producing the predicates body. This is done in one-shot in line 9.

Lastly, the `verify-func` logic begins by checking the well-formedness of the heap-dependent function's pre- and post-conditions in lines 14 and 16, but before checking well-formedness of the post condition we need to `havoc` the return variable in case the post-condition term includes any assertions over the return value. Hence, the symbolic value of the return value needs to be present in $\sigma_2.\gamma$ before producing the post-condition in line 16. After the well-formedness checks for a heap-dependent function's pre/post conditions pass, we then can generate an axiom for the function that will give the SMT solver an interpretation of the function denoted by the syntax `ret` $= id(\overline{args})$ in any later assertions that refer to the function. The details of the axiomatization algorithm which is implemented in the helper `axiom`, is deferred to section 3.7.1 but we give a brief set of details around why we invoke the axiom generation at this point when checking well-formedness. At this point at line 18 we know that the pre- and post-conditions of the heap-dependent function are well-formed and we also have a guarantee that the symbolic states have the symbolic formulas for the pre-condition and the body on the top of $\sigma_2.\pi \in$ PCStack and the symbolic formula for the body (the body of a function is pure at this point by definition of a heap-dependent function) is on the top of the $\sigma_3.\pi \in$ PCStack. Thus, the helper `axiom` can utilize the two states to implement a simple axiom translation algorithm. Once the axiom is generated from these two stacks, the axiom can be propagated to the caller of `verify-func` and later discharged into the SMT stack.

3.6 Predicates

Both GRASShopper and Viper support recursive predicates where an example of a recursive predicate is a list segment,

The support for recursive predicates was directly ported from Viper [24] so we refer the reader to their discussion on evaluating recursive predicates.

```
1 struct Node {
2   var next: Node;
3   var data: Int;
4 }
5
6 predicate lseg(x: Node, y: Node) {
7   x == y ? emp : acc(x.next) &*& acc(x.data) &*& lseg(x.next, y)
8 }
```

Figure 3.13: Definition of a node for a singly-linked list and an `lseg` predicate.

3.7 Heap-dependent Functions

Both GRASShopper and Viper support heap-dependent functions. Heap-dependent functions are pure functions whose formal parameters are arguments with types that correspond to references to objects that are maintained by the heap and do a pure computation. A simple example of a heap-dependent function is a function, `double`, that adds the values carried by two `Node` structures together. Figure 3.14 illustrates a use-case for a heap-dependent function in a client `add_nodes_double` that calls the heap-dependent function `double` and since the pre-condition has an if-then-else expression that requires that `z == x` or `z == y` for each value of `b`, the function will always double the value of `x` or `y`. The post-condition asserts that the return value `r` is always `x.data + z.data` or `y.data + z.data` depending on the truthiness of `b`.

When checking the procedure `add_nodes_double`, the execution of the assignment rule at the call-site of `double` in line 14 will evaluate the right-hand-side of assignment to produce a symbolic function application, $double(true, a, b, a)$, where a and b are the symbolic values of `a` and `b`. In order for the SMT solver to infer that `r >= a.data + a.data` in the post-condition of `add_nodes_double` it must be able to infer that the application of `double` will result in `r >= z.data`. A common way to do this is to axiomatize the function and push the axiom onto the SMT assertion stack.

Here we will illustrate how the GRASShopper symbolic execution engine will generate func-

```
function double(b: Bool, x: Node, y: Node, z: Node) returns (r: Int)
  requires acc(x.data) &*& acc(y.data)
  requires b ? z == x : z == y
  requires x.data > 0 && y.data > 0
  ensures b ? r == x.data + z.data : y.data + z.data
{
  b ? x.data + z.data : y.data + z.data
}

procedure add_nodes_double(a: Node, b: Node) returns (r: Int)
  requires acc(a.data) &*& acc(b.data)
  ensures acc(a.data) &*& acc(b.data) &*& r >= a.data + a.data
{
  var z := double(true, a, b, a);
  r := z;
}
```

Figure 3.14: A heap-dependent function that adds data from two Nodes in a client procedure.

tion axioms for function double in Figure 3.14 from a given symbolic state during the well-formedness check verify_func of Figure 3.12. The goal is to generate an axiom of the form

$$\forall b \in Bool,\ x, y, z \in Loc\langle Node\rangle,\ s_1 \in Snap.$$
$$(b \implies z == x \ \&\&\ fSnap_{Int}(fst^2(snd(s_1))) > 0 \ \&\&$$
$$fSnap_{Int}(fst^2(snd(s_1))) > 0) \implies$$
$$double(b, x, y, z, s_1) = fSnap_{Int}(fst^2(snd(s_1))) + fSnap_{Int}(fst^2(snd(s_1)))$$

After we present details for the axiom generation algorithm for the simpler function double we will discuss more involved example where the heap-dependent function involves recursion.

3.7.1 Axiomatization

The basic idea for generating an axiom is to realize that every axiom has the same form: a universally quantified formula where the binding variables are the formal parameters of double and an additional Snap parameter if a snapshot is referred to in the function body. The body of the

quantified formula is then constructed from the state obtained after producing the pre-condition and then building an *implication* where the pre-condition symbolic formula implies the post-condition and not the body. This is because we assume that the validity of the heap-dependent function has already been checked. However, notice that the quantified formula is not simply the body Form, `r == x.data + z.data`, but rather the return value is replaced by the symbol $double(b, x, y, z, s_1)$.

The algorithm for generating the axiom is defined via $axiom(\sigma_{pre}, \sigma_{post}, id, args, ret) \in$ SymbState×SymbState×Ident×Terms×$Term \nrightarrow$ Form and is essentially a stack diffing approach where we compute the post-condition stack by diffing the state after producing the body and the state after producing the pre-condition,

$$
\begin{aligned}
axiom(\sigma_{pre}, \sigma_{body}, id, args, ret) = \texttt{let } \langle \_, \pi_{body}, \_ \rangle = \sigma_{body} \texttt{ in} \\
\texttt{let } \langle \_, \pi_{pre}, \_ \rangle = \sigma_{pre} \texttt{ in} \\
\texttt{let } \pi_{post} = filter\_unit\_snap(\pi_{body} - \pi_{pre}) \texttt{ in} \\
\texttt{let } s = collect\_snap(\pi_{post}) \texttt{in} \\
\texttt{let } f_{post} = (\bigwedge \pi_{post})[ret/id(args, s)] \texttt{ in} \\
\forall args, s. \bigwedge \pi_{pre} \implies f_{post}
\end{aligned} \tag{3.4}
$$

A few details need further explanation for the axiom definition in Equation 3.4. The first `let` bindings extract the path condition stacks from the symbolic states from producing the body and pre-condition in order to compute the stack for the post-condition. An alternative way to do this is to use the scope identifiers for the PCScopes pushed onto the path condition stack $\pi \in$ PCStack. One could create a fresh scope id for the pre- and post-conditions, and then later scan the PCStacks searching for the corresponding scopes. The stack diffing-based approach was easier to implement and we present this approach due to time constraints.

The helper function $filter\_unit\_snap$ will filter out $s_i == unit$ where $s \in$ Snap constraints that are a result of producing assertions that have pure components in rule 3.3 line 1 to avoid possible inconsistent formulas when folding over π_{pre} and π_{post} to build the conjunctions used in the final axiom formulas. The *collect_snap* helper will scan a stack and collect the snapshot variables that are needed as binders in the universally quantified axiom formula and to be used as last parameters in $id(args, s \ldots)$

3.8 Unfolding Axiom Generation

Perhaps one of the most complex parts of this work was implementing the `unfolding` command in order to support unfolding of recursive predicates that could contain nested if-then-else expressions. This implementation added the `unfolding` expression to the GRASShopper language and essentially implemented the `unfolding` rule from the Viper thesis [24] verbatim. As such we refer the reader to the definition of the `unfolding eval` rule which leverages a complex `join` helper that implements joining of arbitrarily many paths when evaluating nested if-then-else expressions.

Since the implementation follows the Viper implementation verbatim, we will just discuss an example using a heap-dependent function that unfolds a `lseg` predicate and provide a more detailed description of the interplay between axiom generation and the join rule. Consider the following heap-dependent function in Figure 3.15

```
1 function length(x: Node, y: Node) returns (res: Int)
2   requires lseg(x, y)
3   ensures x == y ==> res == 0
4   ensures res >= 0
5 {
6  unfolding lseg(x, y) in x == y ? 0 : 1 + length(x.next, y);
7 }
```

Figure 3.15: Definition of a length function that leverages `unfolding` an `lseg` predicate. For a definition of `lseg` see Figure 3.13.

At the point where the `unfolding` expression is evaluated after the pre-condition is produced, the symbolic state will have an empty stack and $h = pred\langle lseg, [x, y], s\rangle$ since the only assertion in the precondition is `lseg(x, y)`. The first step that will execute is evaluation of of the `unfolding` expression,

```
1 eval(σ1, unfolding lseg(x, y) in x == y ? 0 : 1 + length(x.next, y), Q)
```

The evaluation of `unfolding lseg(x, y) in x == y ? 0 : 1 + length(x.next, y)` will begin with `lseg(x, y)` unfolded so the body of the `lseg(x, y)` predicate will be available in the next state after the transition, and then the `join` rule will begin its work in the continuation Q and will explore each branch of the `in` expression, `x == y ? 0 : 1 + length(x.next, y)`. At the end of exploring each branch, the `join` rule will merge the symbolic states from each branch into a single join function we denote as f_{join}.

At this point the symbolic state will then be a bit complex but can be distilled down to an implication where the `joinFn` is introduced as a variable,

$$\begin{aligned} r =& f_{join} \;\&\& \\ & (x \neq y \implies f_{join} = 0) \;||\; \\ & \quad (x = y \implies f_{join} = 1 + length(fSnap_{Loc}(fst^2(s)), y, snd(s_1))) \end{aligned} \tag{3.5}$$

Now, since the symbolic state contains the result from the join rule Equation 3.4 can be invoked which will generate the axiom,

$$\begin{aligned} &\forall\, x, y \in Loc\langle Node\rangle,\ s, \in \mathsf{Snap},\ f_{join} \in \mathsf{Int}. \\ &\quad length(x,y,s) = f_{join} \;\&\& \\ &\qquad (x \neq y \implies f_{join} = 0) \;||\; \\ &\qquad\quad (x = y \implies f_{join} = 1 + length(fSnap_{Loc}(fst^2(s)), y, snd(s))) \end{aligned} \tag{3.6}$$

Equation 3.6 shows that the main difference between axiom generation when an `unfolding` expression is present is the addition of the join function f_{join} which is handled as a quantified variable, and that the recursion relies on $length(fSnap_{Loc}(fst^2(s)), y, snd(s)))$ being an inductive hypothesis that the SMT solver is able to reason about based on the generated axiom. To ensure that the unfolding doesn't run into an infinite cycle due to the unfolding of mutually recursive predicates we adopted the same mechanism that the Viper implementation used which is to track a list of seen predicate identifiers with a configurable recursion depth. Each unfolding operation will check this global list to see if the currently unfolded predicate has been seen, if so a global counter is incremented, and if the depth is exceeded, we short-circuit the unfolding and produce an uninterpreted function, f_{rec} which is a similar idea to the use of f_{join}. Since this detail is the same as the Viper implementation, we omit details on this discussion.

4 Implementation & Experiments

We present a suite of case-study programs that will be verified using the new GRASShopper symbolic execution engine in order to demonstrate its capabilities and run-times. The overall implementation of the SE engine consists of over 2050 lines of OCaml code split into five modules along with a test suite of 890 lines of GRASShopper programs. The source code is open source and available on GitHub in the GRASShopper repository under the symb-exec-viper branch.

4.1 Linked List Data Structures

Here we present a few examples of GRASShopper programs for verifying algorithms over linked list data structures. These examples will make heavy use of the list segment predicate, `lseg`. The `lseg` predicate will be used in specifications as a shorthand for providing access permissions to segments of a list, and will demonstrate usage of `fold` and `unfold` statements. Recall that Figure 3.13 presents a definition for a typical `Node` and the recursive `lseg` predicate. The first example presented in Figure 4.1 will remove the first node in a linked list and will unfold the `lseg` predicate to emit permissions to the regions for `first.data` and `first.next`.

The symbolic state that is obtained before the `unfold` in line 7 of Figure 4.1 carries a predicate chunk $pred\langle lseg, [v_1, v_2], s_1\rangle$ due to the pre-condition assertion which is consumed by the unfold operation. The unfold operation will then produce the body of the `lseg` predicate which is why the symbolic heap carries access predicates for the `first` data members `first.data` and `first.next`.

```
1  procedure remove_first(first: Node, last: Node)
2    returns (value: Int, second: Node)
3    requires lseg(first, last)
4    requires first != last
5    ensures lseg(second, last)
6  {
7    {σ{γ = {first ↦ v1, second ↦ v2 value ↦ v3 last ↦ v4}; h = [pred⟨lseg, [v1, v2], s1⟩]; π = [v1 ≠ v2]}}
8    unfold lseg(first, last);
9    {σ{γ = {first ↦ v1, second ↦ v2 value ↦ v3 last ↦ v4};
       h = [pred⟨lseg, [fSnap_Loc(fst^3(s1)), v4]⟩, fld⟨data, v1, fst^2(snd(s1))⟩,
                                                    fld⟨next, v1, fst^3(s1)⟩];
                                                    π = [v1 ≠ v2]}}
10
11   value := first.data;
12   second := first.next;
13   {σ{γ = {first ↦ v1, second ↦ v2 value ↦ v3 last ↦ v4};
       h = [pred⟨lseg, [fSnap_Loc(fst^3(s1)), v4]⟩, fld⟨data, v1, fst^2(snd(s1))⟩,
                                                    fld⟨next, v1, fst^3(s1)⟩];
                                  π = [v1 ≠ v2; v3 == fst^2(snd(s1)); v3 == fst^3(s1)]}}
14
15   free first;
16   {σ{γ = {first ↦ v1, second ↦ v2 value ↦ v3 last ↦ v4};
                                  h = [pred⟨lseg, [fSnap_Loc(fst^3(s1), v4]⟩];
                        π = [v1 ≠ v2; v3 == fst^2(snd(s1)); v3 == fst^3(s1)]}}
17
18 }
```

Figure 4.1: A procedure to remove the first node of a linked list with annotations for the symbolic state during the execution.

The assignments to `value` and `second` will first evaluate the right-hand-side expressions to the snapshots corresponding to `first.next` and `first.data` by searching the heap for the respective field chunks and then push equality constraints $v_3 == fst^2(snd(s_1))$ and $v_3 == fst^3(s_1)$ onto the stack. Lastly, the `free` command will remove heap chunks for the all of the declared fields for the receiver `first`, which will result in the heap only carrying the `lseg` predicate chunk in line 16 of Figure 4.1.

In Figures 4.2 and 4.3, we present two additional procedures that will append a node carrying an integer value `v` to the end of a linked list and reverse a linked list, respectively. Since the

```
1 procedure append(lst: Node, v: Int)
2   requires lseg(lst, null) &*& lst != null
3   ensures lseg(lst, null)
4 {
5   unfold lseg(lst, null);
6
7   if (lst.next == null) {
8     var n := new Node();
9     n.data := v;
10    n.next := null;
11    lst.next := n;
12
13    fold lseg(n, null);
14  } else {
15    append(lst.next, v);
16  }
17  fold lseg(lst, null);
18 }
```

Figure 4.2: Procedure to append a node to the end of a linked list.

algorithms are slightly more involved, we omit annotating the symbolic state and will discuss only a few relevant points. Notice that the pre-condition of append in Figure 4.2 at line 2 uses an additional pure constraint that `lst` is non-null. This is necessary so that permissions to `lst.next` in the recursive call can be inferred from unfolding the `lseg` predicate in line 5. This pure formula can be omitted if each time an access predicate is produced, we push an additional disequality constraint onto the stack to denote that the reference is non-null. The `reverse` procedure on line 1 of Figure 4.3 illustrates how list segment predicates can be used in loop invariants to assert that loops that move pointers always guarantee that they are operating over memory regions that are lists. These invariants are necessary so the loop body can contain fold and unfold commands to emit state into the heap in order to verify the code in the loop body.

4.2 Tree Data Structures

Here we present a few simple programs to illustrate the verification of a binary-search tree used to implement a set data structure. The code relies on a `Node` definition and a `tree(r: Node)`

```
1 procedure reverse(lst: Node)
2   returns (rev: Node)
3   requires lseg(lst, null)
4   ensures lseg(rev, null)
5 {
6   rev := null;
7   var curr := lst;
8   unfold lseg(lst, null);
9   while (curr != null)
10    invariant lseg(rev, null)
11    invariant lseg(curr, null)
12    invariant acc(curr.next) &*& curr.next != null
13  {
14    unfold lseg(rev, null);
15    var tmp := curr;
16    curr := curr.next;
17    tmp.next := rev;
18    rev := tmp;
19    fold lseg(rev, null);
20  }
21  fold lseg(lst, null);
22 }
```

Figure 4.3: Procedure to reverse a linked list with an iterative algorithm.

predicate to recursively define the meaning of a tree which are presented in Figure 4.4

The `tree` predicate definition in line 7 of Figure 4.4 defines the base case to be `r == null` as the empty tree, and then for the non-empty case it grants access permissions to the data member and the pointers to the left and right sub-trees. It then states that these pointers point again to memory regions that indeed satisfy the `tree` predicate recursively. We now define a simple procedure to illustrate that the SE engine can reason about recursive predicate calls and demonstrate the `fold` and `unfold` commands which are used in the linked list programs discussed above.

Figure A.10 illustrates a simple recursive program to traverse a binary tree. Much like the linked list programs, we need to unfold and fold the `tree` predicate in order to emit permissions for the left and right sub-trees before recursively calling `traverse`. The unfold command on line 5 of Figure A.10 will evaluate the `root` and then produce access predicate field chunks for `root.data`, `root.left`, and `root.right` so that the pre-condition of the recursive call to `traverse(root.left)` and `traverse(root.right)` will have the proper permissions so that

```
struct Node {
  var left: Node;
  var right: Node;
  var data: Int;
}

predicate tree(r: Node) {
  r == null
  r != null &*& acc(r.data) &*& acc(r.left) &*& acc(r.right)
    &*& tree(r.left) &*& tree(r.right)
}
```

Figure 4.4: Definition of a node for a binary tree and a tree predicate.

the `tree` predicate can be consumed for the left and right sub-trees.

4.3 Results

We present results from running the GRASShopper SE engine on a suite of programs that test the implementation of the operational semantics. The test suite was critical during the implementation of the symbolic execution for exploring the behavior of the rules and for debugging. The results are presented in an evaluation table in Figure 4.1 we label each program with the syntax `tag:procedure_name` where the `tag` denotes the class of program:

- basic - programs that exercise assignments, memory allocation, if-then-else commands, loops, arithmetic expressions, etc.
- frame - programs that test the framing behavior of predicates.
- func - programs that use procedures that use heap-dependent functions.
- list - programs for linked list algorithm verification.
- tree - programs for tree set algorithm verification.

The evaluation table also provides a reference to the source code for each example in the Appendix A and the running times as measured using an Intel Core i7-8565U CPU with 1.80GHz 4 core processor, 32 GB of ram, and the OCaml compiler at version 4.08.1.

Program	Source Code	LoC	Time (ms)
basic:assign	Figure A.1, line 1	6	38
basic:assign2	Figure A.1, line 8	7	19
basic:loop0	Figure A.2, line 5	12	18
basic:loop1	Figure A.2, line 18	16	347
basic:pure1	Figure A.3, line 1	8	17
basic:pure2	Figure A.3, line 10	7	18
basic:pure_swap	Figure A.3, line 18	7	16
basic:assume1	Figure A.3, line 25	7	16
basic:if1	Figure A.4, line 1	10	33
basic:if2	Figure A.4, line 12	10	27
basic:old1	Figure A.4, line 23	9	338
frame:foo_heap	Figure A.5, line 5	6	93
frame:bar_heap	Figure A.5, line 12	8	215
func:add_nodes	Figure A.6, line 13	14	857
func:add_nodes_double_1	Figure A.6, line 31	15	807
list:unfold_lseg	Figure A.7, line 10	9	285
list:fold_left	Figure A.7, line 17	9	272
list:empty_list	Figure A.7, line 24	9	191
list:append	Figure A.8, line 1	22	807
list:append_loop	Figure A.8, line 20	23	288
list:remove_first	Figure A.9, line 1	15	241
list:reverse	Figure A.9, line 14	25	330
tree:traverse	Figure A.10, line 1	11	243
tree:insert	Figure A.11, line 1	32	677
tree:rotate_left	Figure A.12, line 1	20	243
tree:rotate_right	Figure A.12, line 17	20	242

Table 4.1: Evaluation table for GRASShopper programs checked by symbolic execution. The data was collected on a Linux machine running an Intel Core i7-8565U CPU with 1.80GHz 4 core processor, 32 GB of ram, and the OCaml compiler at version 4.08.1. All source code is listed in Appendix A and LoC denotes lines of code.

There are a few notable trends in the running times. Overall, for simple programs the symbolic execution will terminate in the lower millisecond time range, but the more complex the queries

are generated for SMT solver checks the longer the computation time takes. A few programs such as `func:add_nodes` and `func:add_nodes_double_1` approach the 850 millisecond range all involve heap-dependent functions that generate substantial function axiomatization due to branching of if-then-else expressions.

In the future, support to the VCG backend for field-level access permissions, and the elimination of `fold`/`unfold` expressions, plus the support for the translation of `unfolding` expressions would allow for speedup comparisons to be measured for the symbolic execution backend. We leave that to future work, but not all deductive verification systems support VCG generation and symbolic execution so we felt that this detail wasn't strictly necessary for this scope of work.

5 Conclusion

This work demonstrated that the approach taken by Viper and its predecessors is a viable method for building symbolic execution engines for fully automated deductive verification languages that use separation logic. Through the addition of support for injective functions to handle snapshot encoding of structure fields of different types we demonstrated that the semantics of the Viper symbolic execution engine can be adapted to a language with different features.

By going through this process and learning from the existing literature we attempted to improve the formal specification of the interpreter rules to hopefully clarify some hurdles we encountered when implementing heap-dependent function axiomatization. We also provided more detail regarding the relationship between evaluation of unfolding expressions with nested if-then-else expressions and discussed its relationship to axiomatization. Through this, we made a contribution to the implementation of joining arbitrary execution paths in the state-of-the-art.

A Appendix

A.1 Sample Code for Evaluation Results

The code used to generate the evaluation table in Section 4.3 is provided here.

```
procedure assign(x: Int, y: Int)
  requires x == 0 && y == x
  ensures x == 0 && y == x
{
  pure assert x == 0;
}

procedure assign2(x: Int, y: Int)
  requires x == 10 && y == 5
  ensures x == 10 && y == 5
{
  x := 10;
  y := 5;
}
```

Figure A.1: Basic assignment code.

```
struct DNode {
  var data: Int;
}

procedure loop0(x: Int)
  requires x == 0
  ensures x == 1
{
  var i := 1;
  while (i < 2)
    invariant x == i - 1 && i <= 2
  {
    x := i;
    i := i + 1;
  }
}

procedure loop1(x: DNode)
  requires acc(x.data) &*& x.data == 0
  ensures acc(x.data) &*& x.data == 10
{
  var i := 1;
  var y := new DNode();
  y.data := 10;
  while (i < 11)
    invariant acc(x.data) &*& acc(y.data) &*& x.data == i - 1
              &*& y.data == 10 &*& i <= 11
  {
    x.data := i;
    y.data := 10;
    i := i + 1;
  }
}
```

Figure A.2: Basic loop code.

```
procedure pure1(x: Int)
  requires x == 0
  ensures x == 10
{
  x := 2;
  x := x + 5;
  x := x + 3;
}

procedure pure2(x: Int, y: Int)
  requires x == y
  ensures x == y + 8
{
  x := x + 5;
  x := x + 3;
}

procedure pure_swap(x: Int, y: Int)
  requires x == 0 && y == 10
  ensures x == 10 && y == 0
{
  y, x := x, y;
}

procedure assume1(x: Int)
  requires x > 0
  ensures x == 100
{

   x := 10;
   pure assume x == 100;
}
```

Figure A.3: Basic pure assertion code.

```
1 procedure if1(x: Int, y: Int)
2   requires x == 0
3   ensures x > 2
4 {
5   if (x > 2) {
6     x := 3;
7   } else {
8     x := x + 3;
9   }
10 }
11
12 procedure if2(x: Int, y: Int)
13   requires x == 0
14   ensures x >= 3
15 {
16   if (x > 2) {
17     x := 3;
18   } else {
19     pure assume x >= 3;
20   }
21 }
22
23 procedure old1(x: Node, y: Node)
24   requires acc(x.next) &*& acc(y.next)
25   ensures acc(x.next) &*& x.next == old(y.next)
26           &*& acc(y.next) &*& y.next == old(x.next)
27 {
28   var z := x.next;
29   x.next := y.next;
30   y.next := z;
31 }
```

Figure A.4: Basic if-then-else and `old` expression code.

```
1 struct Node {
2   var next: Node;
3 }
4
5 procedure foo_heap(x: Node, y: Node)
6   requires acc(x.next)
7   ensures acc(x.next) &*& x.next == y
8 {
9   x.next := y;
10 }
11
12 procedure bar_heap(x: Node, y: Node)
13   requires acc(x.next) &*& acc(y.next)
14   ensures acc(x.next) &*& acc(y.next) &*& y.next == old(y.next)
15 {
16   var z := y.next;
17   foo_heap(x, y);
18   pure assert (y.next == z);
19 }
```

Figure A.5: Code to illustrate framing of procedure calls.

```
struct Node {
  var data: Int;
  var next: Node;
}

function f(x: Node, y: Node) returns (z: Int)
  requires acc(x.data) &*& acc(y.data)
  ensures x.data + y.data == z
{
  x.data + y.data
}

procedure add_nodes(a: Node, b: Node) returns (c: Node)
  requires acc(a.data) &*& acc(b.data)
  ensures acc(a.data) &*& acc(b.data) &*& acc(c.data) &*& c.data == a.data + b.data
{
  var z := f(a, b);
  c := new Node();
  c.data := z;
}

function double(b: Bool, x: Node, y: Node, z: Node) returns (r: Int)
  requires acc(x.data) &*& acc(y.data)
  requires b ? z == x : z == y
  requires x.data > 0 && y.data > 0
  ensures r >= z.data
{
  b ? x.data + z.data : y.data + z.data
}

procedure add_nodes_double_1(a: Node, b: Node) returns (r: Int)
  requires acc(a.data) &*& acc(b.data)
  ensures acc(a.data) &*& acc(b.data) &*& r == a.data + a.data
{
  var z := double(true, a, b, a);
  r := z;
}
```

Figure A.6: Code to illustrate heap-dependent function calls.

```
struct Node {
  var next: Node;
  var data: Int;
}

predicate lseg(x: Node, y: Node) {
  x == y ? emp : acc(x.next) &*& acc(x.data) &*& lseg(x.next, y)
}

lemma unfold_lseg(x: Node, y: Node)
  requires lseg(x, y) &*& x != y
  ensures acc(x.next) &*& acc(x.data) &*& lseg(x.next, y)
{
  unfold lseg(x, y);
}

lemma fold_left(x: Node, y: Node)
  requires acc(x.next) &*& acc(x.data) &*& lseg(x.next, y) &*& y != x
  ensures lseg(x, y)
{
  fold lseg(x, y);
}

lemma empty_list(x: Node)
  requires emp
  ensures lseg(x, x)
{
  fold lseg(x, x);
}
```

Figure A.7: Code to illustrate fold/unfold verification primitives.

```
1 procedure append(lst: Node, v: Int)
2   requires lseg(lst, null) &*& lst != null
3   ensures lseg(lst, null)
4 {
5   unfold lseg(lst, null);
6
7   if (lst.next == null) {
8     var n := new Node();
9     n.data := v;
10    n.next := null;
11    lst.next := n;
12
13    fold lseg(n, null);
14  } else {
15    append(lst.next, v);
16  }
17  fold lseg(lst, null);
18 }
19
20 procedure append_loop(lst: Node, v: Int)
21   requires lseg(lst, null) &*& lst != null
22   ensures lseg(lst, null)
23 {
24   unfold lseg(lst, null);
25
26   var curr := lst;
27   while (curr.next != null)
28     invariant lseg(lst, curr) &*& acc(curr.next) &*& lseg(curr.next, null)
29   {
30     curr := curr.next;
31     unfold lseg(curr, null);
32   }
33
34   var n := new Node();
35   n.data := v;
36   n.next := null;
37   lst.next := n;
38
39   fold lseg(lst, null);
40 }
```

Figure A.8: Code for appending a node to the end of a linked list

```
1 procedure remove_first(first: Node, last: Node)
2   returns (value: Int, second: Node)
3   requires lseg(first, last)
4   requires first != last
5   ensures lseg(second, last)
6 {
7   unfold lseg(first, last);
8
9   value := first.data;
10  second := first.next;
11  free first;
12 }
13
14 procedure reverse(lst: Node)
15   returns (rev: Node)
16   requires lseg(lst, null)
17   ensures lseg(rev, null)
18 {
19   rev := null;
20   var curr := lst;
21   unfold lseg(lst, null);
22   while (curr != null)
23     invariant lseg(rev, null)
24     invariant lseg(curr, null)
25     invariant acc(curr.next) &*& curr.next != null
26   {
27     unfold lseg(rev, null);
28     var tmp := curr;
29     curr := curr.next;
30     tmp.next := rev;
31     rev := tmp;
32     fold lseg(rev, null);
33   }
34   fold lseg(lst, null);
35 }
```

Figure A.9: Code for removing a node from a linked list and reversing a list.

```
procedure traverse(root: Node)
  requires tree(root);
  ensures tree(root);
{
  unfold tree(root);
  if (root != null) {
    traverse(root.left);
    traverse(root.right);
  }
  fold tree(root);
}
```

Figure A.10: A simple procedure that will recursively traverse a tree.

```
procedure insert(root: Node, value: Int)
  returns (new_root: Node)
  requires tree(root)
  ensures tree(new_root)
{
  unfold tree(root);
  if (root == null) {
    var t := new Node;
    t.left := null;
    t.right := null;
    t.data := value;
    return t;
  } else {
    var n: Node;
    if (root.data > value) {
      n := insert(root.left, value);
      root.left := n;
      return root;
    } else if (root.data < value) {
      n := insert(root.right, value);
      root.right := n;
      return root;
    }
    return root;
  }
  fold tree(root);
  unfold tree(new_root);
}
```

Figure A.11: Insertion of a value into a tree-set.

```
1 procedure rotate_left(h: Node)
2   returns (res: Node)
3   requires tree(h)
4   requires h != null &*& acc(h.right) &*& h.right != null
5   ensures tree(res)
6 {
7   unfold tree(h);
8   var x: Node;
9   x := h.right;
10  h.right := x.left;
11  x.left := h;
12  fold tree(h);
13  unfold tree(res);
14  return x;
15 }
16
17 procedure rotate_right(h: Node)
18  returns (res: Node)
19  requires tree(h)
20  requires h != null &*& acc(h.left) &*& h.left != null
21  ensures tree(res)
22 {
23  unfold tree(h);
24  var x: Node;
25  x := h.left;
26  h.left := x.right;
27  x.right := h;
28  fold tree(h);
29  unfold tree(res);
30  return x;
31 }
```

Figure A.12: A left and right rotation of a tree-set.

Bibliography

[1] A.W. Appel. *Compiling with Continuations*. Online access with purchase: Cambridge Books Online. Cambridge University Press, 1992.

[2] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

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

[4] RM Burstall. Program proving as hand simulation with a little induction. information processing 74. In *Proceedings of IFIP Congress*, volume 74, pages 308–312, 1974.

[5] Bernard Carré and Jonathan Garnsworthy. Spark—an annotated ada subset for safety-critical programming. In *Proceedings of the Conference on TRI-ADA '90*, TRI-Ada '90, page 392–402, New York, NY, USA, 1990. Association for Computing Machinery.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of*

the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

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

[9] R. Hähnle, M. Heisel, W. Reif, and W. Stephan. An interactive verification system based on dynamic logic. In Jörg H. Siekmann, editor, *8th International Conference on Automated Deduction*, pages 306–315, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[10] Reiner Hähnle and Marieke Huisman. *Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools*, pages 345–373. Springer International Publishing, Cham, 2019.

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

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

[13] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *SIGPLAN Not.*, 50(1):637–650, jan 2015.

[14] Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff. Comparing verification condition generation with symbolic execution: An experience report. In Rajeev Joshi, Peter Müller,

and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, pages 196–208, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] K. Rustan M. Leino. This is boogie 2. June 2008.

[16] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. In *FOSAD*, 2009.

[17] Rustan Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference, LPAR-16, Dakar, Senegal*, pages 348–370. Springer Berlin Heidelberg, April 2010.

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

[19] Peter O'Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, jan 2019.

[20] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, page 1–19, Berlin, Heidelberg, 2001. Springer-Verlag.

[21] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper: Complete heap verification with mixed specifications. In *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014 - Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014*, pages 124–139, 2014.

[22] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[23] Malte Hermann Schwerhoff. Symbolic execution for chalice. Master's thesis, ETH Zurich, 2011.

[24] Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verication Using Symbolic Execution.* PhD thesis, ETH Zurich, 2016.

[25] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[26] Jan Smans, Bart Jacobs, and Frank Piessens. Heap-dependent expressions in separation logic. In John Hatcliff and Elena Zucca, editors, *Formal Techniques for Distributed Systems*, pages 170–185, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[27] A. M. Turing. Checking a large routine. *Report of a Conference on High-Speed Automatic Calculating Machines*, pages 67–69, 1949.