Verifying Java Programs with VeriFast

Jan Smans, Bart Jacobs, Frank Piessens, Willem Penninckx, Frédéric Vogels, and Pieter Philippaerts

K.U.Leuven, Belgium

Abstract. VeriFast is a modular, sound program verifier for concurrent Java programs. The verifier takes as input a number of Java source files annotated with method contracts written in a form of separation logic, inductive data type and fixpoint definitions, lemma functions and proof steps. If VeriFast reports that a program is correct, then (1) that program does not raise NullPointer- or ArrayIndexOutOfBoundsExceptions, (2) contains no data races, (3) never violates assertions, and (4) the assumptions described in the method contracts are guaranteed to hold in each execution.
This paper informally introduces the VeriFast features relevant to the verification of Java programs via a number of examples. VeriFast, additional documentation and a large number of example programs are available online at: http://people.cs.kuleuven.be/bart.jacobs/verifast/.

1 Introduction

Reasoning about object-oriented programs is challenging because of aliasing and inheritance. Aliasing makes it hard to perform local reasoning. That is, at each field update one must carefully consider which objects might be affected by the update because the assigned location might be aliased. Inheritance complicates reasoning because the code that is executed for a method call is not selected statically but is instead determined during execution. In particular, one must guarantee that the code that is executed matches the specification used to reason about the method call. Race conditions and deadlocks further exacerbate this challenge in concurrent object-oriented programs.

In the past decade, several reasoning techniques have been proposed to deal with aliasing and inheritance (see Section 9). In this paper, we informally describe the main features of VeriFast, a program verifier for Java[1] based on one such technique called separation logic [3,4]. The remainder of this paper is structured as follows. VeriFast checks that a method satisfies its corresponding method contract via symbolic execution (Section 2). Method contracts can specify the structure of the heap and provide an upper bound on the set of modifiable memory locations via permissions (Section 3). The set of permissions required by a method can be abstracted over via predicates (Section 4). Inductive data types (Section 5), fixpoint functions (Section 6) and lemmas (Section 7)

[1] VeriFast also targets C programs [1,2].

allow developers to describe deep properties of their data structures. In order to reason about inheritance, VeriFast supports dynamically bound instance predicates (Section 8). Finally, we discuss related work (Section 9) and conclude (Section 10).

2 Method Contracts

VeriFast performs *modular* verification. This means that the tool analyzes each method body in isolation using only specifications (not implementations) to reason about method calls. To enforce modular reasoning, VeriFast requires that each method is annotated with a method contract, consisting of a pre- and postcondition. The precondition describes when the method can safely be called, and vice versa the postcondition describes the return value and the effect of the method on the program state. As an example, consider the method `min` of Figure 1. The method's precondition imposes no restrictions, and hence the method can be called at any time with any value for `x` and `y`. The postcondition guarantees that the method's return value, denoted by `result`, is effectively the minimum of `x` and `y`. Note that all specifications are written inside special comments (`/*@ ... @*/`) which are ignored by the Java compiler but recognized by our verifier.

```
int min(int x, int y)
  //@ requires true;
  /*@ ensures result <= x && result <= y &&
              (result == x || result == y); @*/
{
  if(x <= y)
    return x;
  else
    return y;
}
```

Fig. 1. The method `min` annotated with a method contract.

VeriFast checks that a method body complies with the corresponding method contract via symbolic execution [5]. That is, the tool symbolically executes the body starting in a symbolic state that represents an arbitrary concrete state that satisfies the precondition, and verifies that each resulting post-state satisfies the postcondition. A symbolic state consists of three subcomponents: the symbolic store, the path condition and the symbolic heap. The symbolic store is a partial function mapping local variable names to symbolic values. Each symbolic value is a first-order term. The path condition is a list of first-order formulas that encode the assumptions that hold on the path being verified. An inconsistent path condition corresponds to an unreachable state in the program. Finally, the

symbolic heap is a multiset of heap chunks. Each heap chunk has a name and a list of arguments. Both the name and the arguments are first-order terms.

A single symbolic state can represent a large, potentially infinite number of concrete states. For example, the initial symbolic store for the method `min` of Figure 1 maps `x` and `y` to fresh first-order constants. This single symbolic state represents 2^{64} concrete states. A disadvantage of using symbolic values is that the verifier cannot necessarily decide which branch should be taken in a conditional statement or expression. To solve this problem, VeriFast forks the entire symbolic execution at each conditional. For example, the then branch of the if statement in the method `min` - and in general the statements following the conditional - are executed assuming that `x` is less than or equal to `y`, while the else branch is executed with `x > y` added to the path condition.

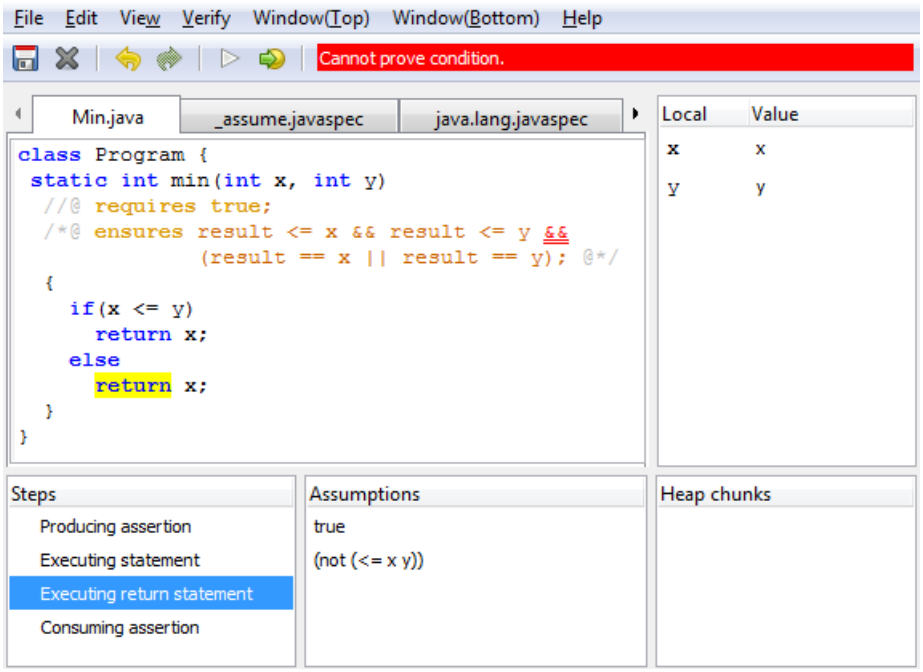

Fig. 2. The VeriFast IDE.

Developers can diagnose verification errors via the VeriFast integrated development environment (IDE) shown in Figure 2. The IDE enables developers to inspect the symbolic states encountered along the path leading to the error in the panel on the bottom left. The path condition, symbolic heap and symbolic store of the currently selected state are respectively displayed in the panels on the bottom center, bottom right and top right.

3 Permissions

An important problem in the verification of imperative programs with aliasing is framing. That is, one should be able to deduce from the method contract an upper bound on the set of memory locations that the corresponding method can modify. VeriFast uses separation logic's permissions [3,4] to tackle the frame problem. More specifically, whenever a method accesses a memory location, it must have permission to do so. `o.f |-> v` denotes (1) the permission to access the field `f` of object `o` and (2) that `o.f` currently holds the value `v`. These permissions are encoded in the symbolic state as heap chunks. As an example, consider the method `shift` in the class `Interval` of Figure 3. `shift`'s precondition requires callers to provide the permissions to access `low` and `high`. The precondition imposes no restrictions on the values of both fields, but binds `l` to the pre-state value of `low` and `h` to the pre-state value of `high`. The postcondition returns both permissions to the caller and states that the lower and upper bounds are shifted by `amount`.

```
class Interval {
  int low, high;

  Interval(int l, int h)
    //@ requires l <= h;
    //@ ensures low |-> l &*& high |-> h;
  { low = l; high = h; }

  void shift(int amount)
    //@ requires low |-> ?l &*& high |-> ?h;
    //@ ensures low |-> l + amount &*& high |-> h + amount;
  { low += amount; high += amount; }

  int getLow()
    //@ requires [?f]low |-> ?l;
    //@ ensures [f]low |-> l &*& result == l;
  { return low; }
}
```

Fig. 3. The class `Interval`.

How do permissions allow developers to deduce frame information? When verifying a method call, the permissions described by the callee's precondition conceptually transfer from the caller to the callee, and vice versa when the call returns, the permissions described by the postcondition transfer from the callee to the caller. If a caller retains the permission to a memory location `o.f` during a call, then the callee cannot modify `o.f` as VeriFast preserves the system invariant that only a single read/write permission exists for `o.f` and only one ac-

tivation record holds that permission. As an example, consider the client code of Figure 4. At program location A, the activation record of `main` holds the permissions to access the fields of `i1` and `i2`. Moreover, the constructor's postcondition holds and hence `i2.low` equals `20`. The precondition of `i1.shift(5)` demands permission to access the fields of `i1` (but not the fields of `i2`). Therefore, `main` retains the permission to `i2.low` and `i2.high` during the call `i1.shift(5)`. As a consequence, we can deduce that `i2.low` still equals `20` at program location *B*, and that the assertion on the next line will succeed in every execution. Note that the proof outlined above depends only on the specification of `Interval`'s methods, not on their implementations.

```
void main() {
  Interval i1 = new Interval(0, 10);
  Interval i2 = new Interval(20, 30); // A
  i1.shift(5); // B
  assert i2.low == 20;
}
```

Fig. 4. Client code for the class `Interval`.

In addition to read/write permissions, VeriFast supports read-only permissions. That is, permissions can be qualified with a fraction between `0` (exclusive) and `1` (inclusive), where `1` denotes full (read and write) permission and any other fraction represents read-only permission. We typically omit writing `[1]` for full permissions. For example, `[f]o.low |-> v` denotes read-only access if `f` is less than `1` and full permission if `f` equals `1`. Permissions can be split and merged as required during the proof. For example, a single read/write permission `[1]o.f |-> v` can be split into two read-only permissions, `[1/2]o.f |-> v` and `[1/2]o.f |-> v`, and vice versa. The contract of the method `getLow` of Figure 3 uses fractions to specify that the method only reads the field `low` by requiring only an arbitrary fraction `f` (implicitly assumed to be non-zero) of that field.

Permissions are useful not only for reasoning about framing, but also to prevent data races. That is, a data race occurs when two threads concurrently access the same memory location and at least one of these accesses is a write operation. VeriFast prevents data races by enforcing the system invariant that for each memory location, the total sum of the fractions of the permissions over all activation records is at most `1`. That is, if an activation record (and by extension the thread containing the activation record) holds the permission to write `o.f`, then no other activation record - in particular activation records in other threads - holds any permission to `o.f`. Thus, if one thread is allowed to write a memory location, then no other thread can concurrently access that memory location. Note that two threads can concurrently read a memory location if both threads hold a read-only permission.

4 Data Abstraction

Data abstraction is crucial in the construction of modular programs, as it ensures that internal changes in a module do not propagate to client code. However, the specification of `Interval` of Figure 3 was not written with data abstraction in mind as the specification exposes the internal fields `low` and `high`. If the internal representation of `Interval` changes, then so does the specification. When the specification of a module changes, client code must be reverified to ensure its correctness.

```
/*@ predicate interval(Interval i, int l, int h) =
  i.low |-> l &*& i.high |-> h &*& l <= h; @*/

class Interval {
  private int low, high;

  Interval(int l, int h)
    //@ requires l <= h;
    //@ ensures interval(this, l, h);
  {
    low = l; high = h;
    //@ close interval(this, l, h);
  }

  void shift(int amount)
    //@ requires interval(this, ?l, ?h);
    //@ ensures interval(this, l + amount, h + amount);
  {
    //@ open interval(this, l, h);
    low += amount; high += amount;
    //@ close interval(this, l + amount, h + amount);
  }

  int getLow()
    //@ requires [?f]interval(this, ?l, ?h);
    //@ ensures [f]interval(this, l, h) &*& result == l;
  {
    //@ open [f]interval(this, l, h);
    return low;
    //@ close [f]interval(this, l, h);
  }
}
```

Fig. 5. An abstract specification for `Interval`.

To abstract over the permissions required by a method, permissions can be grouped and hidden via predicates. For example, consider the predicate `interval` of Figure 5. This predicate groups the permissions to access the fields `low` and `high`, and additionally imposes the constraint that the former field should be less than or equal to the latter. The body of the predicate is visible only in the module defining the class `Interval`; outside of that module, it is simply an opaque container of permissions. As shown in Figure 5, the specification of `Interval` is made implementation-independent by phrasing the effect of its methods in terms of the predicate `interval`. Modifying the internal representation of `Interval` changes the meaning of the predicate `interval`, but does not affect the external specification and hence does not force reverification of client code. Just like basic permissions, predicates can be qualified with a fraction and can be split and merged as required during the proof. For example, the precondition of `getLow` requires an arbitrary fraction `f` of the predicate `interval`.

VeriFast does not automatically fold and unfold predicate definitions (unless the predicate is marked as precise). Instead, developers must explicitly indicate via ghost commands where predicates must be folded and unfolded. For example, the `open` ghost statement in the method `shift` unfolds the definition of the predicate `interval`, and similarly the `close` statement folds the definition.

A predicate is precise if its parameters can be subdivided into in- and output parameters, and the values of the input parameters uniquely determine the values of the output parameters. In VeriFast, a predicate can be marked as precise by placing a semicolon instead of a comma between the input and output parameters. For precise predicates, the verifier can infer certain open and close statements. For example, the parameter `i` of the predicate `interval` can be marked as an input parameter that uniquely determines the values of `l` and `h` by placing a semicolon between the first and second parameter. If `interval` is marked as precise in this way, the open and close statements of Figure 5 can be omitted as VeriFast is able to infer them.

Note that predicates play the role of object invariants. For example, the body of the predicate `interval` specifies that an `Interval` object is consistent if `low` is less than or equal to `high`. The tool does not impose built-in rules that guide when invariants must hold. Instead, developers must explicitly state where invariants are expected to hold by specifying that the validity predicate holds.

5 Inductive Data Types

To allow developers to specify rich properties, VeriFast supports inductive data types. That is, developers can specify the behavior of their imperative data structures by summarizing their state as an instance of an inductive data type. For example, the first line of Figure 6 defines the well-known inductive data type `list`: a list is either empty or the concatenation of a head element and a tail. This definition is generic in the type of the list elements (here `t`). The state of a `Stack` object is summarized as an inductively defined list. That is, the predicate

`stack` defines a correspondence between a `Stack` object `s` and a mathematical list of objects `vs`: `stack(s, vs)` denotes that `s` is a valid stack that contains the objects in the mathematical list `vs`. The predicate `stack` internally uses the recursive predicate `lseg`. The assertion `lseg(from, to, vs)` denotes that there exists a sequence of nodes starting at `from` and ending in `to` that contains the values `vs`. The constructor's postcondition guarantees that `this` is a valid, empty stack. The precondition of `push` requires that `this` is a valid stack, while the postcondition ensures that the target is still a valid stack where `o` is added to the top of the stack.

```
/*@ inductive list<t> = nil | cons(t, list<t>);

predicate lseg(Node from, Node to; list<Object> vs) =
  from == to ?
    vs == nil
  :
    from != null &*& from.value |-> ?v &*&
    from.next |-> ?next &*& lseg(next, to, ?nvs) &*&
    vs == cons(v, nvs);

predicate stack(Stack s; list<Object> vs) =
  s.first |-> ?first &*& lseg(first, null, vs); @*/

class Node { Object value; Node next; ... }

class Stack {
  Node first;

  Stack()
    //@ requires true;
    //@ ensures stack(this, nil);
  {
  }

  void push(Object o)
    //@ requires stack(this, ?vs);
    //@ ensures stack(this, cons(o, vs));
  {
    first = new Node(o, first);
  }
}
```

Fig. 6. The class `Stack` specified via an inductively defined list.

6 Fixpoint Functions

The state of an imperative data structure can be summarized in the specification as an instance of an inductive data type. To allow developers to specify the effect of methods on this state, VeriFast supports fixpoint functions. A fixpoint function is a mathematical function that operates on an inductively defined data type. For example, consider the fixpoint functions defined in Figure 7. The functions `head`, `tail` and `length` respectively return the head, tail and length of a recursively defined list, while `append` concatenates two such lists.

```
/*@
fixpoint t head<t>(list<t> xs) {
  switch (xs) {
    case nil: return default_value<t>;
    case cons(x, xs0): return x;
  }
}

fixpoint list<t> tail<t>(list<t> xs) {
  switch (xs) {
    case nil: return nil;
    case cons(x, xs0): return xs0;
  }
}

fixpoint int length<t>(list<t> xs) {
  switch (xs) {
    case nil: return 0;
    case cons(x, xs0): return 1 + length(xs0);
  }
}

fixpoint list<t> append<t>(list<t> xs, list<t> ys) {
  switch (xs) {
    case nil: return ys;
    case cons(x, xs0): return cons(x, append(xs0, ys));
  }
}
@*/
```

Fig. 7. A number of fixpoint functions.

As shown in Figure 8, the method contracts of `pop` and `size` use fixpoint functions to describe the return value and the effect of the methods on the stack's state. For example, the postcondition of `size` states that the method's return value equals the length of the element list.

Fixpoint functions are encoded as first-order functions in the underlying theorem prover. The behavior of these functions is then encoded via a number of axioms that are derived from the fixpoint's body. To guarantee that these axioms are consistent, we enforce that fixpoint functions terminate. That is, a fixpoint f can call a fixpoint g if either (1) g is defined before f in the program text or (2) the call decreases the size of an inductive parameter.

```
class Stack{
  ...

  Object pop()
    //@ requires stack(this, ?vs) &*& vs != nil;
    /*@ ensures stack(this, tail(vs)) &*&
                result == head(vs); @*/
  {
    Object res = first.value;
    first = first.next;
    return res;
  }

  private int size_helper(Node first)
    //@ requires lseg(first, null, ?vs);
    /*@ ensures lseg(first, null, vs) &*&
                result == length(vs); @*/
  {
    //@ open lseg(first, null, vs);
    if (first == null)
      return 0;
    else
      return 1 + size_helper(first.next);
  }

  int size()
    //@ requires stack(this, ?vs);
    //@ ensures stack(this, vs) &*& result == length(vs);
  {
    return size_helper(this.head);
  }
}
```

Fig. 8. The methods `pop` and `size` specified via fixpoint functions.

7 Lemmas

Lemma functions allow developers to prove properties about their inductive data types, fixpoints and predicates, and allow them to use these properties when reasoning about programs. A lemma is a method without side effects marked `lemma`. The contract of a lemma function corresponds to a theorem, its body to the proof, and a lemma function call to an application of the theorem. VeriFast has two types of lemma functions: pure lemmas and spatial lemmas.

A lemma is pure if its contract does not contain spatial assertions (i.e. basic permissions and predicates). The contract of a pure lemma corresponds to a theorem that states that the precondition implies the postcondition. The function `append_assoc` of Figure 9 is an example of a pure lemma that states that the fixpoint `append` is associative. `append_assoc`'s body proves associativity by induction on `xs`. More specifically, the case `nil` of the switch statement corresponds to the base case, while the case `cons` corresponds to the inductive step. The recursive call in the case `cons` is an application of the induction hypothesis.

```
/*@
lemma void append_assoc<t>(list<t> xs, list<t> ys, list<t> zs)
    requires true;
    ensures append(append(xs, ys), zs) ==
            append(xs, append(ys, zs));
{
  switch(xs) {
    case nil:
    case cons(x0, xs0): append_assoc(xs0, ys, zs);
  }
}

lemma void lseg_merge(Node a, Node b, Node c)
  requires lseg(a, b, ?vs1) &*& lseg(b, c, ?vs2) &*&
           lseg(c, null, ?vs3);
  ensures lseg(a, c, append(vs1, vs2)) &*& lseg(c, null, vs3);
{
  open lseg(c, null, vs3);
  open lseg(a, b, vs1);
  if(a != b) lseg_merge(a.next, b, c)
}
@*/
```

Fig. 9. A pure and a spatial lemma.

Contrary to pure lemmas, spatial lemmas can mention spatial assertions in their method contracts. The contract of a spatial lemma corresponds to a theorem that states that the permissions described by the precondition are equivalent to the permissions described by the postcondition. A spatial lemma does

not modify the values in the heap, but only rewrites the symbolic representation of the symbolic state. `lseg_merge` is an example of a spatial lemma. The corresponding theorem states that a list segment from `a` to `b` and a list segment from `b` to `c` can be merged into a single list segment from `a` to `c`, provided there exists an additional list segment from `c` to `null`.

The body of the iterative implementation of the method `size` shown in Figure 10 calls `lseg_merge` and `append_assoc` to prove that the loop body preserves the loop invariant.

```
class Stack {
  ...

  int size()
    //@ requires stack(this, ?vs);
    //@ ensures stack(this, vs) &*& result == length(vs);
  {
    Node curr = first;
    //@ Node first = curr;
    int count = 0;
    while(curr != null)
      /*@ invariant lseg(first, curr, ?vs1) &*&
                    lseg(curr, null, ?vs2) &*&
                    vs == append(vs1, vs2) &*&
                    count == length(vs1); @*/
    {
      //@ Node oldCurr = curr;
      curr = curr.next;
      //@ open lseg(curr, null, _);
      //@ close lseg(curr, null, _);
      count++;
      //@ lseg_merge(first, oldCurr, curr);
      //@ append_assoc(vs1, cons(head(vs2), nil), tail(vs2));
    }
    //@ open lseg(curr, curr, _);
    return count;
  }
}
```

Fig. 10. An iterative implementation of the method `size`. Lemma calls are needed to prove correctness of this method.

8 Inheritance

A Java interface is a named group of abstract methods. For example, the interface `List` of Figure 11 declares the methods `add`, `get` and `size`. Each non-abstract class that implements this interface must implement these methods.

```
interface List {
  //@ predicate valid(list<Object> vs);

  void add(Object o);
    //@ requires valid(?vs);
    //@ ensures valid(append(vs, cons(o, nil)));

  Object get(int index);
    /*@ requires valid(?vs) &*& 0 <= index &&
                 index < length(vs); @*/
    //@ ensures valid(vs) &*& result == nth(index, vs);

  int size();
    //@ requires valid(?vs);
    //@ ensures valid(vs) &*& result == length(vs);
}
```

Fig. 11. The interface `List` defining the instance predicate `valid`.

The specifications of `add`, `get` and `size` are written in terms of the instance predicate `valid`. An instance predicate is similar to an ordinary predicate, except that (1) it has an implicit this parameter, (2) it has multiple definitions (one for each subclass), and (3) it is dynamically bound on the dynamic type of the implicit argument. As each implementing class has its own fields and invariants, each subclass can assign a different meaning to the predicate. For example, the definition of `valid` in the class `ArrayList` of Figure 12 states that `items` must point to a valid array and that `size` lies between zero and the length of that array. The class `LinkedList` assigns a completely different meaning to `valid`: a linked list is valid if `first` points to a sequence of nodes ending in `null` that contain the values `vs`. Just like methods, instance predicates are dynamically bound: the meaning of the assertion `l.valid(vs)` depends on the dynamic type of `l`. For example, if `l`'s dynamic type is `ArrayList`, then the definition given in the class `ArrayList` is used to interpret `l.valid(vs)`. Note that instance predicates are similar to Parkinson's and Bierman's predicate families [6].

The code that is executed for a method call is not selected statically, but instead depends on the dynamic type of the target object. To guarantee that the code that is selected during execution matches the compile-time contract of the call, VeriFast checks that whenever a subclass implements or overrides a

method, the contract of the overriding and overridden method are specification compatible [7].

```
class ArrayList implements List {
  /*@ predicate valid(list<Object> vs) =
      this.items |-> ?items &*& items != null &*&
      array_slice(items, 0, items.length, ?elems) &*&
      this.size |-> ?size &*& 0 <= size &*&
      size <= items.length &*& vs == take(size, elems); @*/

  Object[] items;
  int size;

  ...
}

class LinkedList implements List {
  /*@ predicate valid(list<Object> vs) =
      this.first |-> ?first &*& lseg(first, null, vs); @*/

  Node first;
  ...
}
```

Fig. 12. The classes `ArrayList` and `LinkedList`. Both classes implement the interface `List` and provide a different implementation for the predicate `valid`.

9 Related Work

Verification of imperative programs with shared, mutable state is an active area of research. In this section, we give a short overview of a number of related tools and approaches.

Separation logic VeriFast reasons about programs using separation logic. Separation logic [3,4] extends classical Hoare logic with a new connective called separating conjunction. A separating conjunction, denoted $P * Q$, holds if both P and Q hold for disjoint parts of the heap. The key proof rule that enables local reasoning is the frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

The frame rule states that C preserves R if that command does not need the permissions described by R.

Berdine, Calcagno, and O'Hearn [5] demonstrate that a fragment of separation logic is amenable to automatic static checking by building a verifier, called

Smallfoot, for a small, procedural language. VeriFast and jStar [8] extend the ideas of Berdine *et al.* to the Java programming language. Unlike VeriFast, jStar can infer certain loop invariants (based on abstraction rules provided by developers in a separate file).

In addition to jStar and VeriFast, Smallfoot has inspired several other tools. For example, Heap-Hop [9] is an extension of Smallfoot targetted at proving memory safety and deadlock freedom of concurrent programs that rely on message-passing. Tuerk [10] has developed HOLFoot, a port of Smallfoot to HOL. He has mechanically proven that HOLFoot is sound. HIP [11] is a variant of Smallfoot that focuses on automatically proving size properties (in addition to shape properties). As it is hard for fully automatic tools to prove full functional correctness, several researchers [12,13,14] have used separation logic within interactive proof assistants.

Leino and Müller [15] and Smans, Jacobs and Piessens [16] have demonstrated that separation logic's key concepts, namely permissions and permission transfer, can be incorporated in an automatic verifier based on first-order verification condition generation and automated theorem proving with an SMT solver [17]. Summers and Parkinson [18] have formalized the relation between separation logic and implicit dynamic frames. An advantage of implicit dynamic frames compared to separation logic is that heap-dependent expressions can be used within specifications and that developers need not provide open and close statements as folding and unfolding is done automatically by the theorem prover. A disadvantage of implicit dynamic frames - and in our experience of other, similar approaches based on verification condition generation and automated theorem proving - is that (1) verification can be slow, (2) it can be hard to diagnose why verification fails and (3) automated theorem proving can be unpredictable, as small changes to the input can cause huge differences in verification time (or even whether the proof succeeds at all).

Dynamic Frames and Regional Logic The dynamic frames [19,20,21,22] approach solves the frame problem by annotating methods with frame annotations. More specifically, the frame annotation declares an upper bound on the set of memory locations that can be read or written by the corresponding method. These upper bounds can be specified using dynamic frames. A dynamic frame is a ghost method or ghost field that returns or holds a set of memory locations. A dynamic frame is *dynamic* in the sense that the set of memory locations represented by a dynamic frame can change over time. A disadvantage of dynamic frames compared to separation logic is that developers must explicitly provide frame annotations. These annotations must be checked by the verifier. Moreover, - to the best of our knowledge - it is not clear how to extend dynamic frames from sequential to concurrent programs.

Dafny [20] and VeriCool [22] are two verifiers based on dynamic frames. Both verifiers rely on verification condition generation instead of symbolic execution.

Java Modeling Language The Java modeling language (JML) [23] is a behavioral interface specification language for Java. An advantage of JML compared to separation logic is that JML specifications can mention Java expres-

sions, thereby making JML easy to learn for developers and amenable to run-time checking. Several tools use JML as their specification language. The extended static checker for Java (ESC/Java) [24] was one of the first tools based on JML. ESC/Java is purposely unsound; the goal of the tool is not to prove full functional correctness, but to find common programming errors such as null dereferences and off-by-one errors. Krakatoa [25] is a JML-based program verifier for Java based on the Why verification platform. Just like VeriFast, Krakatoa has been applied in the verification of Java Card programs. The Key tool [26] is a program verifier based on dynamic logic that uses JML as a specification language. In addition to program verifiers, the JML community has developed run-time assertion checkers and unit test generators [27].

Other Approaches The Verifying C Compiler (VCC) [28] is a program verifier for concurrent C programs. Although the VCC programming methodology is developed specifically for C, we believe many ideas from that methodology can be applied in the verification of object-oriented programs.

10 Conclusion and Future Work

VeriFast is a separation logic-based program verifier for Java. In this paper, we have informally explained the key features of this verifier.

Based on our experience with VeriFast, we identify three main areas of future work. First of all, the basic permissions in separation logic provide either read-only or full access. However, for some programs - in particular concurrent programs - more flexible permissions policies are required. For example, developers should be able to define a permission that permits incrementing (but not decrementing) a memory location. As shown by Owicki and Gries [29] and more recently by Jacobs and Piessens [30], arbitrarily complex policies can be expressed by using ghost state and fractional permissions. However, using ghost state does not lead to intuitive proofs. Therefore, we consider designing a more flexible permission system a key challenge for the future. Concurrent abstract predicates [31] form a promising direction in this area.

VeriFast requires developers to explicitly fold and unfold predicate definitions. Moreover, to prove inductive properties, lemmas must be written and called whenever the property is needed. A second important challenge to be addressed in future work is reducing the annotation overhead by automatically inferring open and close statements and lemma invocations.

Finally, VeriFast supports only a subset of Java. For example, generics are not supported yet by the tool. In order for VeriFast to be applicable to large Java programs, we must extend the supported subset of Java.

References

1. Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2010.

2. Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial. 2011.
3. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002.
4. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic (CSL)*, 2001.
5. Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages (APLAS)*, 2005.
6. Matthew Parkinson and Gavin Bierman. Separation logic, abstraction and inheritance. In *Principles of Programming Languages (POPL)*, 2008.
7. Matthew Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2005.
8. Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2008.
9. Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2009.
10. Thomas Tuerk. A formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.
11. Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *Computer Aided Verification (CAV)*, 2008.
12. Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *International Conference on Functional Programming (ICFP)*, 2008.
13. Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *European Symposium on Programming (ESOP)*, 2008.
14. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *Symposium on Principles of Programming Languages (POPL)*, 2007.
15. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Object-oriented Programming (ESOP)*, 2009.
16. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *European Conference on Object-oriented Programming (ECOOP)*, 2009.
17. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
18. Matthew Parkinson and Alexander Summers. The relationship between separation logic and implicit dynamic frames. In *European Symposium on Programming (ESOP)*, 2011.
19. Ioannis Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods (FM)*, 2006.
20. Anindya Banerjee, David Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, 2008.
21. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2010.

22. Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental approaches to Software Engineering (FASE)*, 2008.
23. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3), 2005.
24. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Languages, Design and Implementation (PLDI)*, 2002.
25. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1 – 2), 2004.
26. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag, 2007.
27. Gary T. Leavens, Yoonsik Cheon, and David R. Cok. Demonstration of JML tools. Technical report, Iowa State University, 2005.
28. Ernie Cohen, Michal Moskał, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *Computer Aided Verification (CAV)*, 2010.
29. Susan Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5), 1976.
30. Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Symposium on Principles of Programming Languages (POPL)*, 2011.
31. Thomas Dinsdale-Young, Mike Dodds, Phillippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *European Conference on Object-oriented Programming (ECOOP)*, 2010.