VeriFast: Imperative Programs as Proofs

Bart Jacobs*, Jan Smans, and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract. We propose an approach for the verification of imperative programs based on the tool-supported, interactive insertion of annotations into the source code. Annotations include routine preconditions and postconditions and loop invariants in a form of separation logic, as well as inductive datatype definitions and recursive function and predicate definitions to enable rich specifications. To enable verification of these rich specifications, annotations also include *lemma routines*, which are like ordinary routines of the programming language, except that it is checked that they do not have side-effects and that they terminate. Recursive lemma routines serve as inductive proofs that their precondition implies their postcondition.
Verification proceeds by symbolic execution, using a separation logic-based representation of memory, and using first-order terms constrained by a first-order theory as symbolic data values. Data value queries are delegated to an SMT solver; since memory framing issues are eliminated from these queries and only well-behaved quantification is used, SMT solver queries perform much better than in verification condition based approaches.
Annotation insertion is supported by an integrated development environment where the user may invoke the verification tool. If verification fails, the user can step through the symbolic execution trace and inspect the symbolic state at each step. Since verification typically takes less than a second, this enables an efficient iterative annotate-and-verify process. Furthermore, it is hoped that by offering proof technology in a form recognizable to programmers, the approach brings interactive program verification to a wider audience.

1 Introduction

In recent years, program verification technology has progressed dramatically, particularly in the area of the abstract specification of a routine's side-effects (known as the heap framing problem), with separation logic [16] emerging as the most promising approach to this problem. More recently still, researchers have built on separation logic to improve the performance and automation of the verification process, by replacing a pure reliance on an SMT solver's rather uninformed quantifier instantiation logic with symbolic execution [4], with a separation logic-based symbolic representation of memory.

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

One of the remaining frontiers of program verification is the integration of this symbolic execution approach with the convenient specification and verification of functional correctness properties, involving not just the shape of data structures, but their contents as well. Existing approaches in this area are either by encoding such properties in first-order logic, to make them acceptable to SMT solvers, but again involving heavy use of potentially ill-behaving quantifiers; or by translating the proof obligation into the input language of an interactive proof assistant, creating a disconnect between the code and the proof.

We propose a novel approach to this problem: we allow the programmer to define inductive datatypes and structural recursive functions over these datatypes inside annotations in the source code, and to inductively prove lemmas about these types and functions by writing *lemma routines* in the target programming language itself.

The datatypes and functions are still axiomatized using quantifiers, and queries about them are still passed to an SMT solver, but by not including exhaustiveness axioms, the SMT solver is kept in check, and queries perform well, while at the same time achieving a very expressive specification language.

The contributions of this paper are as follows:

- We propose a novel verification approach based on separation logic [16], symbolic execution [4] and an SMT solver [8] where simple proof steps can be directly inserted into the source code. Rich properties can be specified via inductive data types and fixpoint functions.
- We propose *lemma routines*, a way of writing proofs as imperative programs: the contract of the lemma routine corresponds to the lemma itself, its body is the proof and a call to a lemma routine corresponds to applying the lemma.
- The VeriFast program verifier for C and Java, an implementation of the above contributions that (1) supports symbolic debugging of specifications and implementations in an IDE and (2) allows for an interactive annotation experience due to the speed and predictability of the verifier.

The remainder of this paper is structured as follows. In Section 2, we describe the building blocks of our verification approach: separation logic, symbolic execution and support for rich specifications via inductive data types and fixpoints. Section 3 extends the specification language with lemma routines. Symbolic debugging of proofs in the VeriFast IDE is then discussed in Section 4. Finally, we compare with related work and conclude in Sections 5 and 6.

2 Building Blocks of the Verification Approach

This section describes the building blocks of our verification approach: separation logic, the combination of symbolic execution and an SMT solver, and supporting rich specifications via inductive data types and fixpoints.

2.1 Separation Logic

The core of our specification language is a form of separation logic. Separation logic [16] is an extension of classical Hoare logic with two new assertions, points-to and separating conjunction. A points-to assertion, $e_1 \rightarrow f \mapsto e_2$, states that the heap contains a memory location at address $\&e_1 \rightarrow f$ with value e_2. A separating conjunction, $P * Q$, states that P and Q hold for disjoint parts of the heap. Separation logic allows for local reasoning via the frame rule:

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

Informally, the frame rule states that the proof for C can ignore the part of the heap that is not accessed by C (here R).

```
struct node { int value; struct node *next; };

predicate node(struct node *n, int value, struct node *next) =
  n→value ↦ value * n→next ↦ next * malloc_block_node(n);

struct node *create_node(int value, struct node *next)
  requires emp;
  ensures node(result, value, next);
{
  struct node *n = malloc(sizeof(struct node));
  if (n == 0) abort();
  n→value = value; n→next = next;
  close node(n, value, next);
  return n;
}
```

Fig. 1. Example demonstrating predicates, contracts and ghost statements.

Let's look at the C program of Figure 1. The program consists of the declaration of the struct *node* and a function for creating new nodes, together with separation logic annotations that describe the behavior of the program. All annotations in this paper are highlighted by a gray background. The first annotation we encounter is a predicate declaration, *node*. The body of the predicate states that the location at address $\&n \rightarrow value$ can be read and written and that the current value of the location is *value*. Similarly, $n \rightarrow next$ points to *next*. The assertion *malloc_block_node*(n) is a predicate assertion that denotes permission to free n. The definition of *malloc_block_node*(n) is not visible in this context. In general, a predicate assertion can be considered a shorthand for the predicate's body with actual parameters substituted for the formal ones.

Each C function is given a corresponding contract, consisting of a precondition P and a postcondition Q. The goal of the verifier is to check that execution of the function's body results in a state satisfying Q, provided the execution started in a state satisfying P. In addition, the verifier checks that the function's body does not perform illegal operations such as dereferencing the 0 pointer. The precondition of *create_node* is **emp**, denoting that the function will not access existing heap locations and that no restrictions apply to the parameters. *create_node*'s postcondition consists of the predicate assertion $node(\mathbf{result}, value, next)$. Recall that predicate assertions are simply shorthands for the predicate's body.

Our verifier does not fold or unfold predicates by itself, but instead developers must explicitly perform such proof steps by inserting ghost commands into the source code. A ghost command is an operation that is ignored during concrete execution, but affects verification. The close statement in the body of *create_node* is an example of such a ghost command that folds the predicate *node*. In particular, this command replaces the body of the predicate by an application of *node*. The converse operation that unfolds a predicate is called open and will be shown in Figure 3.

2.2 Symbolic Execution

We verify whether a program satisfies its specification via symbolic execution along the lines of Berdine *et al.* [4]. Contrary to [4], we use an SMT solver [8] to reason about pure assertions.

Symbolic State A symbolic state is a triple (h, s, Π), consisting of a symbolic heap h, a symbolic store s and a path condition Π. The symbolic heap is a multiset of heap chunks, where each chunk consists of a predicate name [1] and a list of first-order terms (one for each predicate parameter). The symbolic store is a partial function from variable names to terms. Finally, the path condition is a set of first-order formulas that describe restrictions over the logical symbols used in the symbolic state that hold on the current execution path.

Symbolic Execution The rules for symbolic execution are defined in continuation-passing-style in Figure 2. In particular, the continuation Q represents the work that remains to be done on the current path. Note that for brevity, only some of the rules are shown.

The rules for produce and consume represent respectively assuming and proving a separation logic assertion. For example, producing a (spatial) points-to assertion $e \rightarrow f \mapsto ?x$ amounts to adding a new field chunk to the heap whose value is a fresh first-order symbol and binding x to that symbol. (More generally, a question mark preceding an variable in an assertion indicates a binding occurrence of the variable.) Producing a pure assertion, such as $e_1 = e_2$, corresponds

[1] The VeriFast implementation uses a logical term instead of a literal predicate name in order to support higher-order predicates.

to adding the equivalent formula to the path condition. Producing a conditional assertion $e_1 = e_2 ? A_1 : A_2$ creates two branches in the symbolic execution, where the condition is added to Π and A_1 is produced on the first branch, and the negation of the condition is added to Π and A_2 is produced on the other branch. Note that a particular branch is not reachable if the path condition is inconsistent. The rules check reachability by checking consistency when adding an assumption.

$$
\begin{array}{l}
\mathsf{produce}(h, s, \Pi, e{\rightarrow}f \mapsto ?x, Q) \text{ where } e \text{ is of type } \textbf{struct } S * \equiv \\
\quad \textbf{let } \sigma = \mathsf{fresh}(h, s, \Pi) \textbf{ in } Q(h \uplus \{S\_f([\![e]\!]_s, \sigma)\}, s[x := \sigma], \Pi) \\
\\
\mathsf{produce}(h, s, \Pi, e_1 = e_2, Q) \equiv \Pi \not\vdash_{\mathrm{SMT}} \neg[\![e_1 = e_2]\!]_s \Rightarrow Q(h, s, \Pi \cup \{[\![e_1 = e_2]\!]_s\} \\
\\
\mathsf{produce}(h, s, \Pi, A_1 * A_2, Q) \equiv \\
\quad \mathsf{produce}(h, s, \Pi, A_1, (\lambda h, s, \Pi.\ \mathsf{produce}(h, s, \Pi, A_2, Q))) \\
\\
\mathsf{produce}(h, s, \Pi, e_1 = e_2 ? A_1 : A_2, Q) \equiv \\
\quad (\Pi \not\vdash_{\mathrm{SMT}} \neg[\![e_1 = e_2]\!]_s \Rightarrow \mathsf{produce}(h, s, \Pi \cup \{[\![e_1 = e_2]\!]_s\}, A_1, Q)) \wedge \\
\quad (\Pi \not\vdash_{\mathrm{SMT}} [\![e_1 = e_2]\!]_s \Rightarrow \mathsf{produce}(h, s, \Pi \cup \{\neg[\![e_1 = e_2]\!]_s\}, A_2, Q)) \\
\\
\mathsf{consume}(h, s, \Pi, e{\rightarrow}f \mapsto ?x, Q) \text{ where } e \text{ is of type } \textbf{struct } S * \equiv \\
\quad \exists t_1, t_2, h'.\ h = \{S\_f(t_1, t_2)\} \uplus h' \wedge \Pi \vdash_{\mathrm{SMT}} t_1 = [\![e]\!]_s \wedge Q(h', s[x := t_2], \Pi) \\
\\
\mathsf{consume}(h, s, \Pi, e_1 = e_2, Q) \equiv \Pi \vdash_{\mathrm{SMT}} [\![e_1 = e_2]\!]_s \wedge Q(h, s, \Pi) \\
\\
\mathsf{consume}(h, s, \Pi, A_1 * A_2, Q) \equiv \\
\quad \mathsf{consume}(h, s, \Pi, A_1, (\lambda h, s, \Pi.\ \mathsf{consume}(h, s, \Pi, A_2, Q))) \\
\\
\mathsf{consume}(h, s, \Pi, e_1 = e_2 ? A_1 : A_2, Q) \equiv \\
\quad (\Pi \not\vdash_{\mathrm{SMT}} \neg[\![e_1 = e_2]\!]_s \Rightarrow \mathsf{consume}(h, s, \Pi \cup \{[\![e_1 = e_2]\!]_s\}, A_1, Q)) \wedge \\
\quad (\Pi \not\vdash_{\mathrm{SMT}} [\![e_1 = e_2]\!]_s \Rightarrow \mathsf{consume}(h, s, \Pi \cup \{\neg[\![e_1 = e_2]\!]_s\}, A_2, Q)) \\
\\
\mathsf{verify}(h, s, \Pi, f(e), Q) \text{ where } f(x) \textbf{ requires } A_1;\ \textbf{ensures } A_2;\ \equiv \\
\quad \mathsf{consume}(h, \{(x, [\![e]\!]_s)\}, \Pi, A_1, (\lambda h, s', \Pi. \\
\qquad \mathsf{produce}(h, s', \Pi, A_2, (\lambda h, \_, \Pi.\ Q(h, s, \Pi))))) \\
\\
\mathsf{valid}(f(x) \textbf{ requires } A_1;\ \textbf{ensures } A_2;\ \{\ \overline{s}\ \}) \equiv \\
\quad \textbf{let } \sigma = \mathsf{fresh}(\emptyset, \emptyset, \emptyset) \textbf{ in} \\
\quad \mathsf{produce}(\emptyset, \{(x, \sigma)\}, \emptyset, A_1, (\lambda h, s, \Pi. \\
\qquad \mathsf{verify}(h, \{(x, \sigma)\}, \Pi, \overline{s}, (\lambda h, \_, \Pi. \\
\qquad\quad \mathsf{consume}(h, s, \Pi, A_2, (\lambda h, \_, \_.\ h = \emptyset))))))
\end{array}
$$

Fig. 2. Symbolic execution

Consumption is the inverse operation. For example, consumption of a points-to assertion looks for a heap chunk that matches the assertion. If none exists, consumption fails; otherwise, the chunk is removed.

The role of production and consumption of assertions is illustrated in the verification rule for function calls (verify), where the precondition is consumed and

the postcondition is produced. Notice that variables bound in the precondition are in scope in the postcondition.

Finally, a function declaration is valid if after producing the precondition with arbitrary values for the function parameters and symbolically executing the function's body in the resulting symbolic state, the postcondition can successfully be consumed. The final continuation ($\lambda h, \_, \_.\ h = \emptyset$) enforces the absence of memory leaks by checking that the heap is empty after consuming the postcondition. A program is valid if all function declarations are valid.

2.3 Inductive Data Types

To allow for rich specifications, we support inductive data types in our annotation language. As an example, consider the code of Figure 3 with functions for creating and adding elements to a linked list. The first annotation in the program is the inductive data type *list* which is defined in the traditional way: a list is either empty or the combination of a head element[2] and a tail.

The inductive data type is used in the predicates *lseg* and *list*. $lseg(n1, n2, vs)$ holds if there exists a chain of next pointers starting at *n1* and ending in *n2*, where the values in the nodes along the chain correspond to *vs*. Similarly, $list(l, vs)$ holds if l is a pointer into a list data structure that contains *vs* as elements. Note that the definitions of the above predicates need not be visible to client code.

The contracts of *create_list* and *add_to_front* are defined in terms of the predicate *list*. The former function has no precondition and creates a empty list, while the latter adds a new element x to the front of the existing list. The parameter *?vs* in the precondition of *add_to_front* represents an existential variable that is bound at the time the function is called.

Encoding Each constructor of an inductive data type is encoded as a function symbol in the SMT solver. Values of an inductive data type are then encoded as applications of the corresponding function. For example, the inductive data type *list* has two corresponding functions[3]:

$$nil : inductive$$
$$cons : int \times inductive \rightarrow inductive$$

Inductive data type constructors satisfy (1) disjointness, (2) injectiveness, and (3) exhaustiveness. Property (1) states that the ranges of the constructors are disjoint. Property (2) states that the constructors are injective. These two properties are encoded as axioms in the SMT solver. Exhaustiveness, on the other hand, is not available to the SMT solver to prevent arbitrary case splits. VeriFast only performs the case splits called for explicitly in the source code using **switch** statements.

[2] The inductive data type in Figure 3 specifically uses **int** as the element type. However, the VeriFast implementation supports generic inductive definitions. For simplicity, we use the non-generic version in this paper.

[3] We use the same SMT solver type for all inductive data types.

```
inductive list = nil | cons(int, list);
struct list { struct node *first; struct node *last; };

predicate lseg(struct node *n1, struct node *n2, list vs) =
  n1 == n2 ? vs == nil : node(n1, ?h, ?next) * lseg(next, n2, t) * vs == cons(h, t);
predicate list(struct list *l, list vs) =
  l→first ↦?fn * l→last ↦?ln * lseg(fn, ln, vs) * node(ln, _, _) * malloc_block_list(l);

void *create_list()
  requires emp;
  ensures list(result, nil);
{
  struct list *l =
    malloc(sizeof(struct list));
  if (l == 0) abort();
  struct node *n =
    create_node(0, 0);
  l→first = n; l→last = n;
  close lseg(n, n, nil);
  close list(l, nil);
  return l;
}

void add_to_front(struct list *l, int x)
  requires list(l, ?vs);
  ensures list(l, cons(x, vs));
{
  open list(l, vs);
  struct node *n =
    create_node(x, l→first);
  l→first = n;
  close list(l, cons(x, vs));
}
```

Fig. 3. Example demonstrating inductive data types.

2.4 Fixpoints

In addition to inductive data types, our specification language also includes fixpoint functions. A fixpoint function is a function with at least one argument of an inductive type. The body of a fixpoint function must be a **switch** statement over one of these arguments, called its *inductive argument*. The function *length* in Figure 4 is an example of a fixpoint that computes the length of an inductively defined list. To ensure that fixpoints are well-defined, the body of a fixpoint f can use another fixpoint g only if g appears before f in the program text or if g equals f and one of the components of the inductive argument is used in the place of that argument. The fixpoint *length* is used in the contracts of *list_length* and *list_length_helper* to relate the state of the list to the result of those functions.

Encoding Each fixpoint function is encoded as a function symbol in the SMT solver. Applications of fixpoints are encoded as applications of the corresponding function. For each case in the switch statement in the body of the fixpoint function, an axiom is generated that enables the solver to "compute" the value of the fixpoint for the corresponding case. For example, two axioms are generated

for the fixpoint *length*:

$$length(nil) = 0$$
$$\forall h, t \bullet \{length(cons(h,t))\}\ length(cons(h,t)) = 1 + length(t)$$

The expression $\{length(cons(h,t))\}$ represents a trigger [9] that ensures that the axiom is well-behaved.

```
fixpoint int length(list l) {
  switch(l) {
    case nil : return 0;
    case cons(h, t) :
      return 1 + length(t);
  }
}
```

```
int list_length_helper
  (struct node *n1, struct node *n2)
  requires lseg(n1, n2, ?vs);
  ensures lseg(n1, n2, vs) * result == length(vs);
{
  open lseg(n1, n2, vs);
  if (n1 == n2) return 0;
  else return 1 + list_length_helper(n1→next);
  close lseg(n1, n2, vs);
}

int list_length(struct list *l)
  requires list(l, ?vs);
  ensures list(l, vs) * result == length(vs);
{
  open list(l, vs);
  return list_length_helper(l→first, l→last);
  close list(l, vs);
}
```

Fig. 4. Example demonstrating fixpoints.

Note that our specification library already includes several inductive definitions and fixpoints that can be reused in many applications.

Quantifiers Our specification language does not include quantifiers. The main reason for avoiding general quantifiers (and only internally generating quantifiers for fixpoints and inductive definitions) is that they lead to poor performance. However, certain forms of quantification can be encoded easily using fixpoints or recursive predicates. For example, given the predicate *forall* defined below, separation logic's *iterated star* over a list $\circledast_{x \in l}\ p(x)$ can be written as $forall(l, p)$, meaning $p(x)$ holds separately for each element x of list l.

```
predicate forall⟨α⟩(list⟨α⟩ l, predicate(α) p) =
  switch(l) {
    case nil : return true;
    case cons(h, t) : return p(h) * forall(t, p);
  };
```

Here α is a type parameter and p ranges over predicates with one argument of type α, making *forall* a higher-order predicate. Note that *forall* must be folded and unfolded explicitly using ghost commands.

3 Programs as Proofs

The verification approach described in the previous section lacks the power to prove certain properties. For example, the property that the length of an inductively defined list is never negative is crucial for proving that the assert statement in *my_function* never fails:

```
void my_function(struct list * l)
  requires list(l, ?vs); ensures list(l, vs);
{ int len = list_length(l); assert(0 <= len); }
```

However, as it stands there is no way to prove this property. In particular, the SMT solver cannot deduce the property as it does not perform induction and furthermore it does not have access to the exhaustiveness property.

To allow developers to prove properties such as the one described above, we introduce lemma routines, a way of writing theorems and proofs as C functions directly in the source code. A lemma routine is a C function annotated with the keyword **lemma**. The contract of the function is a theorem stating that the precondition implies the postcondition for all possible values of the function's parameters. For example, the lemma routine *length_nonnegative* of Figure 5 states that the length of an inductively defined list is never negative.

```
lemma void length_nonnegative(list l)
  requires true;
  ensures 0 <= length(l);
{
  switch(l) {
    case nil : break;
    case cons(h, t) :
      length_nonnegative(t); break;
  }
}
```

```
void my_function(struct list * l)
  requires list(l, ?vs);
  ensures list(l, vs);
{
  int len = list_length(l);
  length_nonnegative(vs);
  assert(0 <= len);
}
```

Fig. 5. Example demonstrating pure lemma functions.

The body of a lemma routine represents a proof of the theorem. For the proof to be valid, the body must satisfy certain restrictions. First of all, the body must

not affect the concrete state; specifically, it must not perform field updates or call regular C functions. Secondly, execution of the body must terminate. We enforce termination by disallowing loops and by restricting what lemmas the body can call. More specifically, a lemma routine x can call a lemma routine y provided y appears before x in the program text or x equals y. If the lemma performs a recursive call (i.e. x equals y), then one of the following restrictions must hold:

1. The recursive call decreases the size of the heap. More specifically, after consuming the precondition of the recursive call, a field chunk must remain in the heap.
2. The recursive call decreases the size of an inductive parameter. More specifically, the body of the lemma is a switch statement over an inductive parameter, and one of the components of the inductive parameter is used in the place of the inductive argument in the recursive call.
3. The recursive call decreases the derivation depth of the first conjunct of the precondition. More specifically, the body of the lemma is not a switch statement, and the first chunk consumed by the precondition of the recursive call was obtained by unfolding the first chunk produced by the precondition of the lemma.

The proof of *length_nonnegative* is by induction on the structure of the list l. The recursive call in the proof is allowed because it is applied to a component of l (restriction 2 described above). Note that checking the proof simply consists of symbolically executing the lemma routine as a normal C function and checking that the additional restrictions described above are satisfied. Also, the proof goes through because of the SMT solver's built-in theory of linear arithmetic.

A call of a lemma routine corresponds to an application of the theorem for the arguments of the call. For example, the body of *my_function* in Figure 5 calls *length_nonnegative* with argument *vs*, thereby instantiating the theorem for *vs*. Since *len* equals *length*(*vs*), the verifier can prove the assert statement never fails. Note the lemma routine call is a ghost command that does not affect the execution of the actual C program. Moreover, from the point of view of the caller, the lemma routine is just another function whose precondition must be established and whose postcondition can be assumed.

The contract of *length_nonnegative* contains only pure assertions. However, lemma routine contracts are allowed to use spatial assertions, as illustrated by the lemmas *distinct_nodes* and *add_lemma* of Figure 6. The former lemma states that having two *node* predicates in the symbolic heap implies that the node pointers themselves are different. The latter lemma states that a heap containing a list segment from *n1* to *n2*, a node object at *n2* with next pointer *n3* and a node at *n3* is equivalent to a heap containing just a list segment from *n1* to *n3* and a node at *n3*. The proof of the latter lemma runs by induction on the size of the heap. *add_lemma* is needed to prove correctness of the C function *add_to_back* in Figure 6.

```
fixpoint list add(list l, int x) {
  switch(l) {
    case nil : return cons(x, nil);
    case cons(h, t) : return cons(h, add(t, x));
  }
}

lemma void distinct_nodes(struct node * n1, struct node * n2)
  requires node(n1, ?n1v, ?n1n) * node(n2, ?n2v, ?n2n);
  ensures node(n1, n1v, n1n) * node(n2, n2v, n2n) * n1 ≠ n2;
{
  open node(n1, n1v, n1n); open node(n2, n2v, n2n);
  close node(n1, n1v, n1n); close node(n2, n2v, n2n);
}

lemma void add_lemma(struct node * n1)
  requires lseg(n1, ?n2, ?vs) * node(n2, ?v, ?n3) * node(n3, _, _);
  ensures lseg(n1, n3, add(vs, v)) * node(n3, _, _);
{
  distinct_nodes(n2, n3); open lseg(n1, _, _);
  if (n1 == n2) {
    close lseg(n3, n3, nil);
  } else {
    distinct_nodes(n1, n3); open node(n1, ?n1v, ?n1n);
    add_lemma(n1→next); close node(n1, n1v, n1n);
  }
  close lseg(n1, n2, add(vs, v));
}

void add_to_back(struct list * l, int x)
  requires list(l, ?vs); ensures list(l, add(vs, x));
{
  open list(l, vs);
  struct node * n = create_node(0, 0); struct node * nl = l→last;
  open node(nl, _, _); nl→next = n; nl→value = x; close node(nl, x, n);
  l→last = n; add_lemma(l→first); close list(l, add(vs, x));
}
```

Fig. 6. Example demonstrating spatial lemma functions

Lemmas can play the role of public invariants [15]. In particular, a lemma routine in a header file represents a restriction for the implementor of that header file, and an assumption for client code.

4 Symbolic Debugging with VeriFast

The verification approach described above has been implemented in a verifier prototype for C and Java called VeriFast. Builds for Windows, Mac and Linux are available at `http://www.cs.kuleuven.be/~bartj/verifast`.

4.1 Symbolic Debugging

The combination of an integrated development environment (IDE) that supports symbolic debugging and fast feedback on verification attempts leads to an interactive specification and verification experience. The IDE, shown in Figure 7, displays not only the code itself, but also the contents of the symbolic state at each step to allow developers to debug failed verification attempts. In particular, the box on the bottom left shows all intermediate symbolic states encountered during symbolic execution of the program. The user can select a symbolic state, and inspect the symbolic store (top right), the chunks in the symbolic heap (bottom right) and the path condition (bottom center).

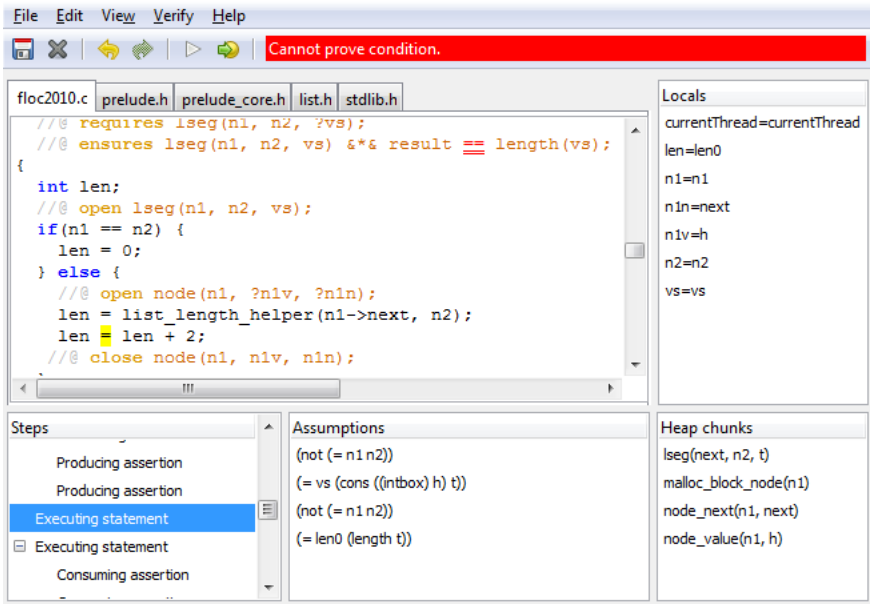

Fig. 7. The VeriFast IDE.

VeriFast has been used to prove correctness of a number of C and Java programs, including a small chat server, a composite design pattern [13], a kernel-like program illustrating VeriFast's support for unloadable modules and fine-grained concurrent algorithms. All verified programs are included in the Verifast distribution. Table 1 shows a number of verified programs.

program	total # lines	# annotation lines	time taken (seconds)
chat server	242	114	0.08
linked list and iterator	332	194	0.09
composite	345	263	0.09
JavaCard applet	340	95	0.51
GameServer	383	148	0.23

Table 1. Overview of a number of programs verified using VeriFast.

4.2 Automation

VeriFast supports a number of techniques to automate proofs: auto lemmas and automatic folding and unfolding of predicates.

Auto lemmas Pure lemmas (i.e. lemmas whose contract does not contain spatial assertions) can be annotated with the keyword **auto**. Such lemmas are axiomatized in the theorem prover as $\forall x_1, \ldots, x_n \bullet P \Rightarrow Q$, where $x_1, \ldots x_n$ are the lemma's parameters, and P and Q are respectively its pre and postcondition. Auto lemmas are applied automatically by the theorem prover and need not be called explicitly by the developer. The lemma *length_nonnegative* of Figure 5 is an example of a pure lemma that can be automated. Note: auto lemmas are intended for use by experts only; inappropriate use can cause SMT solver non-termination.

Automatic folding and unfolding When, during consumption, a chunk is not found, and the corresponding predicate is marked specially, VeriFast considers auto-folding and auto-unfolding. Specifically, it looks for a chunk in the heap that contains the missing chunk, or one that is contained by the missing chunk. In the former case, it will perform an unfold; in the latter case, a fold. Note that if the fold fails, or the chunk is still not found, no backtracking is performed, so performance and diagnosability are not adversely affected. The automatic fold and unfold operations end up in the trace and are therefore visible to the developer inspecting a failed proof attempt. Automatic folding and unfolding reduces the number of open and close statements needed for the examples in this paper from 22 to 8.

4.3 Teaching

VeriFast has been used in courses on software engineering and program verification at K.U.Leuven and ETH Zürich. A tutorial on VeriFast covering design

by contract, basic separation logic, loop invariants, predicates, inductive data types, fixpoints, lemmas, overflow checking, function pointers, higher-order predicates, predicate families, fractional permissions and concurrency is available at `http://www.cs.kuleuven.be/~bartj/verifast`.

5 Related Work

To the best of our knowledge, VeriFast is the first tool that enables proofs of rich properties of real programs through source code annotations.

Existing verification tools that take real program source files as input are VCC [7] and Frama-C [2], taking C sources; ESC-Java [11], KeY [3], Jahob [18], and jStar [10], taking Java sources; and Spec# [1], taking C# sources. Of these tools, jStar is the only one that operates using symbolic execution with a separation logic representation of memory. The other tools are based either on Dijkstra's weakest preconditions, treating the heap as an array-valued global variable (VCC, Frama-C, ESC-Java, and Spec#), or on a combination of weakest preconditions and an interactive proof assistant (KeY and Jahob).

jStar's verification approach is in many ways similar to that of VeriFast. A major difference is that jStar attempts to infer loop invariants through abstraction, whereas VeriFast requires loop invariants to be specified explicitly. jStar's abstraction algorithm is driven by user-specified rules, which require significant expertise to author, and whose soundness is not checked by the tool. In fact, jStar does not include any mechanism for performing manual proofs. Furthermore, diagnosing failed verification attempts in jStar is significantly more challenging than in VeriFast due to the reliance on backtracking search.

The weakest preconditions-based approach generates verification conditions (VCs) in first-order logic and sends them to an SMT solver such as Simplify or Z3. This approach suffers from poor performance and predictability, due mainly to the large number of universally quantified premises included in the VCs to deal with the framing of heap effects; the performance of SMT solvers on such queries varies greatly. The situation gets worse if one attempts to express rich properties in these systems, since, due to the absence of inductive datatypes or fixpoint functions, this requires quantification over e.g. the indices of an array or the elements of a set of graph nodes.

To overcome these limitations, KeY and Jahob provide the option of falling back to an interactive proof development mode. In KeY, program proofs in dynamic logic can be constructed graphically using a special-purpose GUI; however, these proofs do not become part of the source code. In Jahob, one can prove lemmas in Isabelle or Coq. This suffers from the problem that these tools are generic proof assistants and are not optimized for proving separation logic properties of imperative programs.

Ynot [6] is a powerful system for separation logic-based verification of imperative programs written in an imperative-functional language shallowly embedded (using monads) in the term language of the Coq proof assistant. Although the programs must be written in this ML-like Coq-embedded language, which is

significantly less convenient that a standalone programming language, a tool is provided that translates these programs into ML or Haskell. Proofs may use all of the power of Coq, but strong tactics are provided that automate common proof steps.

Larch [12] is an approach for the formal specification of program modules. In the Larch approach, a module specification is in the form of a module signature of the programming language, annotated with preconditions and postconditions written in the Larch Behavioral Interface Specification Language (BISL). In these annotations, one can refer to mathematical types and operators defined in the Larch Shared Language (LSL), including inductive datatypes and recursive functions over them. The Larch system includes the Larch Prover, a proof assistant for proving theorems about LSL theories; however, the system is for specification only; verification is outside its scope, and no verification tool is provided.

The L4.verified project [14, 17] proved the correspondence of a microkernel implemented in C with a high-level abstract specification of the microkernel's complete behavior. To verify that the model and the C code have the same behavior, they parsed the C code into an Isabelle/HOL embedding of C, and then interactively proved the correspondence theorem in the Isabelle/HOL proof assistant. While proving properties using VeriFast is more convenient than through the interface of a proof assistant, the property proved by the L4.verified project is stronger than what can currently be expressed conveniently in VeriFast's specification language.

The specification and proof approach centered around inductive definitions was adopted from the Coq proof assistant [5].

VeriFast is powered by the excellent SMT solver Z3 [8].

6 Conclusion

We propose a program verification approach that allows rich properties to be verified through interactive insertion of annotations into the source code. Annotations include inductive datatypes, structural-recursive functions, and recursive spatial predicates for expressiveness, lemma routines for proof power, preconditions and postconditions for specification, and loop invariants and ghost commands for aiding the proof. The verification approach facilitates diagnosis of failed verification attempts, and, combined with its good performance, enables a convenient interactive annotation insertion experience.

We are validating the approach in various application areas, including fine-grained concurrent data structures, device drivers, and smart card applets; initial results are encouraging.

The primary area of future work is in increasing the degree of automation, by investigating ways to infer ghost commands, loop invariants, routine contracts, and lemma applications and implementations.

References

1. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.
2. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language.
3. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
6. Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.
7. Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, 2009.
8. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
9. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3), 2005.
10. Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
11. Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.
12. John V. Guttag, James J. Horning with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
13. Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the composite pattern using separation logic. In *SAVCBS*, 2008.
14. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.
15. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Department of Computer Science, Iowa State University, 2005.
16. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
17. Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In *TPHOLs*, 2009.
18. Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.