Annotation Inference for Separation Logic Based Verifiers

Frédéric Vogels, Bart Jacobs, Frank Piessens, Jan Smans

Annotation inference for separation logic based verifiers

Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans*

Katholieke Universiteit Leuven, Leuven, Belgium
{frederic.vogels,bart.jacobs,frank.piessens,jans.smans}@cs.kuleuven.be

Abstract. With the years, program complexity has increased dramatically: ensuring program correctness has become considerably more difficult with the advent of multithreading, security has grown more prominent during the last decade, etc. As a result, static verification has become more important than ever.

Automated verification tools exist, but they are only able to prove a limited set of properties, such as memory safety. If we want to prove full functional correctness of a program, other more powerful tools are available, but they generally require a lot more input from the programmer: they often need the code to be verified to be heavily annotated.

In this paper, we attempt to combine the best of both worlds by starting off with a manual verification tool based on separation logic for which we develop techniques to automatically generate part of the required annotations. This approach provides more flexibility: for instance, it makes it possible to automatically check as large a part of the program as possible for memory errors and then manually add extra annotations only to those parts of the code where automated tools failed and/or full correctness is actually needed.

1 Introduction

During the last decade, program verification has made tremendous progress. However, a key issue hindering the adoption of verification is that a large amount of annotations is required for tools to be able to prove programs correct, in particular if correctness involves not just memory safety, but also program-specific properties. In this paper, we propose three annotation inference and/or reduction techniques in the context of separation logic-based verifiers: (1) automatic predicate folding and unfolding, (2) predicate information extraction lemmas and (3) automatic lemma application via shape analysis. All aforementioned contributions were developed in the context of the VeriFast program verifier in order to reduce annotation overhead for practical examples.

VeriFast [17] is a verification tool being developed at the K.U. Leuven. It is based on separation logic [21] and can currently be used to verify a multitude of correctness-related properties of C and Java programs. Example usages are

* Jan Smans is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

- ensuring that C code does not contain any memory-related errors, such as memory leaks and dereferencing dangling pointers;
- checking that functions or methods satisfy contracts describing their intended semantics;
- preventing the occurrence of data races in multi-threaded code.

VeriFast heavily relies on programmer-provided annotations: this makes the tool very efficient, but the need for annotations can make its use quite cumbersome. To give the reader an idea of the quantity of annotations needed, we provide some quick statistics in Figure 1: the first column contains a count of the number of lines of actual C code. The second column expresses the number of annotations in number of lines. The numbers between parentheses correspond to the number of open and close statements, respectively, which will be further explained in Section 3.1. The third column shows the amount of annotation overhead.

	LOC	**LOAnn**	**LoAnn/LOC**
stack (C)	88	198 (18/16)	2.3
sorted binary tree (C)	125	267 (16/23)	2.1
bank example program (C)	405	127 (10/22)	0.31
chat server (C)	130	114 (20/26)	0.88
chat server (Java)	138	144 (19/28)	1.0
game server (Java)	318	225 (47/63)	0.71

Fig. 1. Some line count statistics

We have developed three different techniques to (partially) automate verification by mechanizing the generation of (some of) the necessary annotations. In this paper, we describe these three approaches in detail. We will make the distinction between two layers:

- VeriFast's core, which requires all annotations and performs the actual verification. This core must be as small and uncomplicated as possible, as the verification's soundness relies on it.
- The automation layer, which generates as large a portion of the necessary annotations as possible, which will then in a second phase be fed to VeriFast's core for verification.

This approach maximizes robustness: we need only trust the core and can freely experiment with different approaches to automation without having to worry about introducing unsound elements, since the generated annotations will still be fully verified by the core, thus catching any errors.

In order to be able to discuss and compare our three annotation generation techniques, we need the reader to be familiar with VeriFast, therefore we included a small tutorial (Section 2) which explains the basic concepts. Next, we explain our different approaches to automation in Section 3. We then put them side by side in Section 4 by comparing how many of the necessary annotations they are able to generate.

2 VeriFast: a quick tutorial

This section contains a quick introduction to VeriFast. It is not our intention to teach the reader how to become proficient in using VeriFast, but rather to provide a basic understanding of certain concepts on which VeriFast is built. A full tutorial is available at [1].

2.1 A singly linked list

```
struct list { struct list* next; int value; };
struct list* new(struct list* n, int v)
 //@ requires emp;
 /*@ ensures malloc_block_list(result) &*& result->next  |-> n &*&
             result->value |-> v       &*& result != 0         @*/
{
 struct list* node = malloc( sizeof( struct list ) );
 if ( node == 0 ) abort();
 node->next = n; node->value = v;
 return node;
}
```

Fig. 2. A singly linked list node in C

Figure 2 shows a `struct`-definition for a singly linked list node together with a function `new` which creates and initializes such a node. In order to verify this function in VeriFast, we need to provide a contract. The precondition, `emp`, indicates that the function does not require anything to be able to perform its task. The postcondition is a separating conjunction (`&*&`) of the "heap fragments":

- `malloc_block_list(result)` means that the returned pointer will point to a malloc'ed block of memory the size of a `list`. It is produced by a call to `malloc` and it is meant to be eventually consumed by a matching `free`. Failure to do so eventually leads to an unreachable `malloc_block_list` which corresponds to a memory leak.
- `result->next |-> n` means two things: it grants the permission to read and write to the `node`'s `next` field, and it tells us that the `next` field contains the value of the argument `n`. Idem for `result->value |-> v`.
- `result != 0` guarantees that the returned pointer is not null.

If this function verifies, it will mean that it is both memory safe and functionally correct. We defer the explanation of the separating conjunction `&*&` until later.

VeriFast uses symbolic execution [7] to perform verification. The precondition determines the initial symbolic state, which in our case is empty. VeriFast then proceeds by symbolically executing the function body.

1. `malloc` can either fail or succeed. In the case of `list`, its contract is

   ```
   requires emp;
   ensures  result == 0 ? emp
                        : malloc_block_list(result) &*&
                          result->next  |-> _       &*&
                          result->value |-> _;
   ```

   Either it will return `0`, which represents failure, and the heap is left unchanged. Otherwise, it returns a non-null pointer to a block of memory the size of a `struct list`, guaranteed by `malloc_block_list(result)`. It also provides access to both `next` and `value` fields of this newly created node, but does not make any promises about their values.
2. The `if` statement will intercept execution paths where `malloc` has failed and calls `abort`, causing VeriFast not to consider these paths any further.
3. Next, we assign `n` to the `next` field. This operation is only allowed if we have access to this field, which we do since the previous `if` statement filtered out any paths where allocation failed, and a successful `malloc` always provides the required access rights. This statement transforms the `result->next |-> _` heap fragment to `result->next |-> n`.
4. Assigning `v` to `node->value` works analogously.
5. Finally, we return the pointer to the node. At this point, VeriFast checks that the current execution state matches the postcondition. This is the case, which concludes the successful verification of `new`.

A full explanation of the separating conjunction `&*&` can be found in [21]. In short, `P &*& Q` means that the heap consists of two disjoint subheaps where `P` and `Q` hold, respectively. It is used to express that blocks of memory do not overlap, and therefore changes to one object do not influence another. Separating conjunction enables the frame rule, which allows us to reason "locally". It expresses the fact that if an operation behaves some way in a given heap (`{P} op {Q}`), it will behave the same way in an extended heap (`{P &*& R} op {Q &*& R}`). For example, if we were to call `malloc` twice in a row, how can we be sure that it didn't return the same block of memory twice? We know this since, thanks to the frame rule, we get two `malloc_block_list` heap fragments joined by a separating conjunction, guaranteeing us that we are dealing with two different objects.

2.2 Predicates

As shown in the previous section, freely working with a single node requires carrying around quite a bit of information. This can become tiresome, especially if one considers larger structures with more fields whose types might be other structures. For this reason, VeriFast provides the ability to perform abstractions

```
predicate Node(struct list* n, struct list* m, int v) =
 malloc_block_list(n) &*& n->next |-> m &*& n->value |-> v &*& n != 0;

struct list* new(struct list* n, int v)
 //@ requires emp;
 //@ ensures Node(result, n, v);
{
 struct list* node = malloc( sizeof( struct list ) );
 if ( node == 0 ) abort();
 node->next = n; node->value = v;
 //@ close Node(node, n, v);
 return node;
}
```

Fig. 3. Predicates

using predicates [20] which make it possible to "fold" (or close, in VeriFast terminology) multiple heap fragments into one.

Figure 3 shows the definition for the `Node` predicate and an updated version of the `new` function. The `close` statement removes three heap fragments (`malloc_block_list` and the two field access permissions) and replaces them by a single `Node` fragment, on condition that it can ascertain that `node` is not `0` and the three fragments are present on the symbolic heap, otherwise verification fails. This closing is necessary in order to have the final execution state match the postcondition. Closing must happen last, for otherwise the field access permissions would be hidden when they are needed to initialize the node's fields (third line of the procedure body.) At a later time, whenever the need arises to access one of a node's fields, the `Node` fragment can be opened up (replacing `Node` by the three separate heap fragments on the symbolic heap) so as to make the field access permission available again.

2.3 Recursive predicates

A linked list consists of a chain of nodes each pointing to the next. Currently, we can only express linked lists of fixed maximum length:

```
p == 0 ? emp
       : Node(p, q, v1) &*& (q == 0 ? emp
                                    : Node(q, 0, v2)) // len 0-2
```

We can solve this problem using recursive predicates: we need to express that a list is either empty or a node pointing to another list. Figure 4 shows the definition for `LSeg(p, q, xs)`, which stands for a cycleless singly linked list segment where `p` points to the first node, `q` is the one-past-the-end node and `xs` stands for the contents of this list segment.

Figure 4 also contains the definition for a `prepend` function. The contract fully describes its behaviour, i.e. that a new element is added in front of the list,

```
/*@ inductive List = Nil | Cons(int, List);
    predicate LSeg(struct list* p, struct list* q, List Xs) =
    p == q ? Xs == Nil
           : Node(p,?t,?y) &*& LSeg(t,q,?Ys) &*& Xs == Cons(y,Ys); @*/

struct list* prepend(struct list* xs, int x)
 //@ requires LSeg(xs, 0, ?Xs);
 //@ ensures LSeg(result, 0, Cons(x, Xs));
{
 struct list* n = new(xs, x);
 //@ open Node(n, xs, x);
 //@ close Node(n, xs, x);
 //@ close LSeg(n, 0, Cons(x, Xs));
 return n;
}
```

Fig. 4. Recursive predicates

and that the pointer passed as argument becomes invalid; instead the returned pointer must be used.

2.4 Lemmas

The reader might wonder why a `Node` is consecutively opened and closed in Figure 4. Let us first examine what happens without it. When VeriFast reaches the closing of the `LSeg`, execution forks due to the conditional in the `LSeg` predicate:

- `n` might be equal to `0`, in which case `xs` must be `Nil` instead of `Cons(x, Xs)`, so that closing fails. This needs to be prevented.
- `n` could also be a non-null pointer: the `Node` and original `LSeg` get merged into one larger `LSeg`, which is exactly what we want.

Therefore, we need to inform VeriFast of the fact that `n` cannot be equal to `0`. This fact is hidden within the `Node` predicate; opening it exposes it to VeriFast. After this we can immediately close it again in order to be able to merge it with the `LSeg` heap fragment.

The need to prove that a pointer is not null occurs often, and given the fact that an `open`/`close` pair is not very informative, it may be advisable to make use of a lemma, making our intentions clearer, as shown in Figure 5. In Section 3, we will encounter situations where lemmas are indispensable if we are to work with recursive data structures.

3 Automation techniques

We have implemented three automation techniques which we discuss in the following sections. For this, we need a running example: Figure 7 contains a fully

```
/*@ lemma void NotNull(struct list* p)
      requires Node(p, ?pn, ?pv);
      ensures  Node(p, pn, pv) &*& p != 0;
    {
      open Node(p, pn, pv); close Node(p, pn, pv);
    }
@*/
struct list* prepend(struct list* xs, int x)
 //@ requires LSeg(xs, 0, ?Xs);
 //@ ensures  LSeg(result, 0, Cons(x, Xs));
{
 struct list* n = new(xs, x);
 //@ NotNull(n);
 //@ close LSeg(n, 0, Cons(x, Xs));
 return n;
}
```

Fig. 5. The `NotNull` lemma

annotated list-copying function, for which we will try to automatically infer as many of the required annotations as possible.

In our work, we have focused on verifying memory safety; automation for verifying functional properties is future work. Therefore, we simplify the `Node` and `LSeg` predicates we defined earlier by having the predicates throw away the data-related information, as shown in Figure 6.

```
predicate Node(struct list* P, struct list* Q) =
 P != 0 &*& malloc_block_list(P) &*& P->next  |-> Q &*& P->value |-> ?v;

predicate LSeg(struct list* P, struct list* Q) =
 P == Q ? emp : Node(P, ?R) &*& LSeg(R, Q);
```

Fig. 6. Simplified `Node` and `LSeg` predicates

While it is not strictly necessary to understand the code in Figure 7, we choose to clarify some key points:

- The `new()` function produces a new `Node(result, 0)` and always succeeds. It is just another function defined in terms of `malloc` and `abort`s on allocation failure (comparable to Figure 3).
- `NoCycle`, `Distinct`, `AppendLSeg` and `AppendNode` are lemmas whose contracts are shown in Figure 8.

The `copy` function comprises 12 statements containing actual C code, while the annotations consist of 31 statements, not counting the lemmas since these can be shared by multiple function definitions. We now proceed with a discussion of how to generate some of these annotations automatically.

3.1 Auto-open and auto-close

As can be seen in the examples, a lot of annotations consist of opening and closing predicates. This is generally true for any program: the values between parentheses in Figure 1 indicate how many open and close statements respectively were necessary for the verification of other larger programs.

These annotations seem to be ideal candidates for automation: whenever the execution of a statement fails, the verifier could take a look at the current execution state and try opening or closing predicates to find out whether the right heap fragments are produced.

For example, assume we are reading from the `next` field of a variable `x`, which requires a heap fragment matching `x->next |-> v`. However, only `Node(x, y)` is available. Without automation, verification would fail, but instead, the verifier could try opening `Node(x, y)` and find out that this results in the required heap fragment. Of course, this process could go on indefinitely given that predicates can be recursively defined. Therefore, some sort of heuristic is needed to guide the search.

We have added support for automatic opening and closing of predicates [20] to VeriFast. Without delving too much into technical details, VeriFast keeps a directed graph whose nodes are predicates and whose arcs indicate how predicates are related to each other. For example, there exists an arc from `LSeg` to `Node` meaning that opening an `LSeg` yields a `Node`. However, this depends on whether or not the `LSeg` does represent the empty list. To express this dependency, we label the arcs with the required conditions. These same conditions can be used to encode the relationships between the arguments of both predicates. For the predicate definitions from Figure 6, the graph would contain the following:

$$\begin{tikzcd}\mathrm{LSeg}(a,b) \arrow[r, "\begin{array}{c} a \neq b \\ a = p \end{array}"] & \mathrm{Node}(p,q) \arrow[r, "p = x"] & x \rightarrow \mathrm{next} \mapsto y\end{tikzcd}$$

When, during verification, some operation requires the presence of a `Node(p, q)` heap fragment but which is missing, two possible solutions are considered: we can either attempt to perform an auto-open on an `LSeg(p, b)` for which we know that `p != b`, or try to close `Node(p, q)` if there happens to be a `p->next |-> ?` on the current heap.

Using this technique yields a considerable decrease in the amount of necessary annotations: each `open` or `close` indicated by `// a` is inferred automatically by VeriFast. Out of the 31 annotation statements, 17 can be generated, which is more than a 50% reduction.

3.2 Autolemmas

We now turn our attention to another part of the annotations, namely the lemmas. On the one hand, we have the lemma definitions. For the moment, we

```
struct list* copy(struct list* xs)
 //@ requires LSeg(xs, 0);
 //@ ensures LSeg(xs, 0) &*& LSeg(result, 0);
{
 if ( xs == 0 ) {
  //@ close LSeg(0, 0);                             // a
  return 0; }
 else {
  struct list* ys = new();
  //@ open LSeg(xs, 0);
  //@ open Node(xs, _);                             // a
  //@ open Node(ys, 0);                             // a
  ys->value = xs->value;
  struct list *p = xs->next, *q = ys;
  //@ close Node(ys, 0);                            // a
  //@ close Node(xs, p);                            // a
  //@ NoCycle(xs, p);
  //@ close LSeg(p, p);                             // a
  //@ close LSeg(xs, p);                            // a
  //@ close LSeg(ys, q);                            // a
  while ( p != 0 )
   //@ invariant LSeg(xs,p) &*& LSeg(p,0) &*& LSeg(ys,q) &*& Node(q,0);
  {
   //@ struct list *oldp = p, *oldq = q;
   struct list* next = new();
   //@ Distinct(q, next);
   //@ open Node(q, 0);                             // a
   q->next = next; q = q->next;
   //@ close Node(oldq, q);                         // a
   //@ open LSeg(p, 0);
   //@ assert Node(p, ?pn);
   //@ NoCycle(p, pn);
   //@ open Node(p, _);                             // a
   //@ open Node(q, 0);                             // a
   q->value = p->value; p = p->next;
   //@ close Node(q, 0);                            // a
   //@ close Node(oldp, p);                         // a
   //@ AppendLSeg(xs, oldp); AppendNode(ys, oldq);
  }
  //@ open LSeg(p, 0);                              // a
  //@ NotNull(q);                                   // b
  //@ close LSeg(0, 0);                             // a
  //@ AppendLSeg(ys, q);
  //@ open LSeg(0, 0);                              // a
  return ys;
 }
}
```

Fig. 7. Copying linked lists

```
lemma void NoCycle(struct list* P, struct list* Q)
  requires Node(P, Q) &*& LSeg(Q, 0);
  ensures  Node(P, Q) &*& LSeg(Q, 0) &*& P != Q;
lemma void Distinct(struct list* P, struct list* Q)
  requires Node(P, ?PN) &*& Node(Q, ?QN);
  ensures  Node(P, PN)  &*& Node(Q, QN) &*& P != Q;
lemma void AppendLSeg(struct list* P, struct list* Q)
  requires LSeg(P, Q) &*& Node(Q, ?R) &*& Q != R &*& LSeg(R, 0);
  ensures  LSeg(P, R) &*& LSeg(R, 0);
lemma void AppendNode(struct list* P, struct list* Q)
  requires LSeg(P, Q) &*& Node(Q, ?R) &*& Node(R, ?S);
  ensures  LSeg(P, R) &*& Node(R, S);
```

Fig. 8. Lemmas

have made no efforts to automate this aspect as lemmas need only be defined once, meaning that automatic generation would only yield a limited reduction in annotations.

On the other hand we have the lemma applications, which is where our focus lies. Currently, we have only implemented one very specific and admittedly somewhat limited way to automate lemma application. While automatic opening and closing of predicates is only done when the need arises, VeriFast will try to apply all lemmas regarding a predicate `P` each time `P` is produced, in an attempt to accumulate as much extra information as possible. This immediately gives rise to some obvious limitations:

- It can become quite inefficient: there could be many lemmas to try out and many matches are possible. For example, imagine a lemma operates on a single `Node`, then it can be applied to every `Node` on the heap, so it is linear with the number of `Node`s on the heap. If however it operates on two `Node`s, matching becomes quadratic, etc. For this reason, two limitations are imposed: lemmas need to be explicitly declared to qualify for automatic application, and they may only depend on one heap fragment.
- Applying lemmas can modify the execution state so that it becomes unusable. For example, if the `AppendLSeg` lemma were applied indiscriminately, `Node`s would be absorbed by `LSeg`s, effectively throwing away potentially crucial information (in this case, we “forget” that the list segment has length 1.) To prevent this, autolemmas are not allowed to modify the symbolic state, but instead may only extend it with extra information.

Given these limitations, in the case of our example, only one lemma qualifies for automation: `NotNull`. Thus, every time a `Node(p, q)` heap fragment is added to the heap, be it by closing a `Node` or opening an `LSeg` or any other way, VeriFast will immediately infer that `p != 0`. Since we only needed to apply this lemma once, we decrease the number of annotations by just one line (Figure 7, indicated by `// b`).

3.3 Automatic shape analysis

Ideally, we would like to get rid of all annotations and have the verifier just do its job without any kind of interaction from the programmer. However, as mentioned before, the verifier cannot just guess what behaviour a piece of code is meant to exhibit, so that it can only check for things which are program-independent bugs, such as data races, dangling pointers, etc.

Our third approach for reducing annotations focuses solely on shape analysis [13], i.e. it is limited to checking for memory leaks and invalid pointers dereferences. Fortunately, this limitation is counterbalanced by the fact that it is potentially able to automatically generate all necessary annotations for certain functions, i.e. the postcondition, loop invariants, etc.

In order to verify a function by applying shape analysis, we need to determine the initial program state. The simplest way to achieve this is to require the programmer to make his intentions clear by providing preconditions. Even though it appears to be a concession, it has its advantages. Consider the following: the function `length` requires a list, but `last` requires a non-empty list. How does the verifier make this distinction? If `length` contains a bug which makes it fail to verify on empty lists, should the verifier just deduce it is not meant to work on empty lists?

We could have the verifier assume that the buggy `length` function is in fact correct but not supposed to work on empty lists. The verification is still sound: no memory-related errors will occur. A downside to this approach is that the `length` function will probably be used elsewhere in the program, and the unnecessary condition of non-emptiness will propagate. At some point, verification will probably fail, but far from the actual location of the bug. Requiring contracts thus puts barriers on how far a bug's influence can reach.

One could make a similar case for the postconditions: shape analysis performs symbolic execution and hence ends up with the final program state. If the programmer provides a postcondition, it can be matched against this final state. This too will prevent a bug's influence from spreading.

Our implementation of shape analysis is based on the approach proposed by Distefano et al. [13]. The idea is simple and very similar to what has been explained earlier in Section 3.1: during the symbolic execution of a function, it will open and close the predicates as necessary to satisfy the precondition of the operations it encounters. However, the analysis has a more thorough understanding of the lemmas: it will know in what circumstances they need to be applied. A good example of this is the inference of the loop invariant where shape analysis uses the lemmas to abstract the state, which is necessary to prevent the symbolic heap from growing indefinitely while looking for a fixpoint. Consider the following pseudocode:

$$p' := p; \textbf{while } p \neq 0 \textbf{ do } p := p \rightarrow \text{next } \textbf{end}$$

Initially, the symbolic heap contains `LSeg(p, 0)`. To enter the loop, `p` needs to be non-null, hence it is a non-empty list and can be opened up to `Node(p', p1) &*& LSeg(p1, 0)`. During the next iteration, `p1` can be null (the loop ends) or non-null (a second node). Thus, every iteration adds the possibility of an extra node. This way, we’ll never find a fixed point. Performing abstraction will fold nodes back into `LSeg`s. The difference is shown in Figure 9. One might wonder why the abstraction doesn’t also merge both `LSeg`s into a single `LSeg`. The reason for this is that the local variable `p` points to the start of the second `LSeg`: folding would throw away information deemed important.

For our purposes, the algorithms defined in [13] need to be extended so that apart from the verification results of a piece of code and final program states which determine the postcondition, they also generate the necessary annotations to be added to the verified code. This way, the results can be checked by VeriFast, keeping our trusted core to a minimum size (i.e. we do not need to trust the implementation of the shape analysis tool), and extra annotations can be added later on if we wish to prove properties other than memory safety.

without abstraction	**with abstraction**
`Node(p’, p) &*& LSeg(p, 0)`	`LSeg(p’, p) &*& LSeg(p, 0)`
`Node(p’, p1) &*& Node(p1, p) &*& LSeg(p, 0)`	`LSeg(p’, p) &*& LSeg(p, 0)`

Fig. 9. Finding a fixed point

For our example, shape analysis is able to deduce all `open` and `close` annotations, the lemma applications, the loop invariant and the postcondition (in our implementation, we chose to require only the precondition and we manually check that the generated postcondition is as intended). Hence, the number of necessary annotations for Figure 7 is reduced to 1, namely the precondition.

4 Comparison

C-code	**#code**	**A**	**B**	**C**	**D**	**lemma**	**A**	**B**	**C**
`length`	10	12	9	9	1	`Distinct`	9	7	7
`sum`	11	11	7	7	1	`NotNull`	7	6	6
`destroy`	9	6	4	4	1	`AppendNode`	19	16	16
`copy`	23	32	15	14	1	`AppendLSeg`	27	19	18
`reverse`	12	9	5	5	1	`AppendNil`	9	7	6
`drop_last`	28	28	13	13	1	`NoCycle`	11	10	9
`prepend`	7	5	3	3	1				
`append`	13	20	11	11	1				

	#code	**A**	**B**	**C**	**D**
total	113	205	132	128	8

Fig. 10. Annotation line count comparison

In order to get a better idea of by how much we managed to decrease the number of annotations, we wrote a number of list manipulation functions. There are four versions of the code:

(A) A version with all annotations present.
(B) An adaptation of (A) where we enabled auto-open and auto-close.
(C) A version where we take (B) and make `NotNull` an autolemma (Section 3.2).
(D) Finally, a minimal version with only the required annotations to make our shape analysis implementation (Section 3.3) able to verify the code.

Figure 10 shows how the annotation line counts relate to each other.

5 Related work

Smallfoot [6] is a verification tool based on separation logic which given pre- and post-conditions and loop invariants can fully automatically perform shape analysis. It has been extended for greater automation [24], for termination proofs [8, 12], fine-grained concurrency [10] and lock-based concurrency [15].

jStar [14] is an another automatic separation logic based verification tool which targets Java. One only needs to provide the pre- and post-conditions for each method, after which it attempts to verify it without extra help. It is able to infer loop invariants, for which it uses a more generalized version of the approach described by Distefano et al. [13]. This is achieved by allowing the definition of user-defined rules (comparable to our lemmas) which are then used by the tool to perform abstraction on the heap state during the fixed point computation.

A third verification tool based on separation logic is SpaceInvader [13, 24], which performs shape analysis on C programs. Abductor, an extension of this tool, uses a generalized form of abduction [9], which gives it the ability not only to infer loop invariants and postconditions, but also preconditions.

Other tools which don't rely on separation logic are for example KeY [3] (dynamic logic [16]), Spec♯ [5], Chalice [19], Dafny [2], and VCC [11], the latter three being based on Boogie2 (verification condition generation [4, 18]). Still other alternatives to separation logic are implicit dynamic frames [23] and matching logic [22], the latter being an approach where specifications are expressed using patterns which are matched against program configurations.

6 Conclusion

We can divide verifiers in two categories.

- Fully automatic verifiers which are able to determine whether code satisfies certain conditions without any help of the programmer. Unfortunately, this ease of use comes with a downside: these tools can only check certain properties for certain patterns of code. More ambitious verifications such as ensuring full functional correctness remains out of the scope of these automatic verifiers, since correctness only makes sense with respect to a specification, which needs to be provided by the programmer.

- Non-automatic tools are able to perform more thorough verifications (such as full functional correctness), but these require help from the programmer.

In practice, given a large body of code, it is often sufficient to check only automatically provable properties except for a small section of critical code, where a proof of full functional correctness is necessary. Neither of the above two options is then ideal. Our proposed solution is to combine the best of both worlds by using the following verification framework: at the base lies the non-automatic "core" verifier (in our case VeriFast), which will be responsible for performing the actual verification. To achieve this, it requires code to be fully annotated, but in return, it has the potential of checking for a wide variety of properties. On this base we build an automation layer, consisting of specialized tools able to automatically verify code for specific properties. Instead of just trusting the results of these tools, we require them to produce annotations understood by the core verifier.

A first advantage is that only the core verifier needs to be trusted. Indeed, in the end, all automatically produced annotations are fed back to the core verifier, so that unsoundnesses introduced by buggy automation tools will be caught.

A second advantage is that it allows us to choose which properties are checked for which parts of the code. For example, in order to verify a given program, we would start with unannotated code, on which we would apply an automatic verification tool, such as the shape analysis tool discussed in Section 3.3. This produces a number of annotations, which are fed to the core verifier. If verification succeeds, we know the application contains no memory-related errors.

Now consider the case where a certain function `foo` appears to be troublesome and shape analysis fails to verify it, which could mean that all other parts of the code which call this function also remain unverified. In order to deal with this problem the programmer can manually add the necessary annotations for `foo`, let the core verifier check them, and then re-apply the shape analysis tool, so that it can proceed with the rest of the code.

After the whole program has been proved memory-safe, one can proceed with the critical parts of the code where a proof of full functional correct is required. Thus, it makes an iterative incremental approach to verification possible where manually added annotations aid the automatic tools at performing their task.

In this paper, we presented preliminary experience gained in our work in progress towards this goal. Future work includes gaining additional experience with larger programs, gaining experience with the usability of an iterative infer-annotate process, and improving the power of the inference algorithm.

Acknowledgments This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. `http://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf`

2. Dafny: An Automatic Program Verifier for Functional Correctness. In: LPAR-16 (2010)
3. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling 4(1) (2005)
4. Barnett, M., Chang, B.E., Deline, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO (2006)
5. Barnett, M., Leino, Schulte, W.: The Spec# Programming System: An Overview (2005)
6. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2005)
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic Execution with Separation Logic. In: APLAS (2005)
8. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: POPL (2008)
9. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
10. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: In SAS. LNCS. The MIT Press (2007)
11. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOLs. pp. 23–42 (2009)
12. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: POPL '07 (2007)
13. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis based on Separation Logic. In: In TACAS (2006)
14. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA (2008)
15. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: APLAS (2007)
16. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Philosophical Logic (1984)
17. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: APLAS (2010)
18. Leino, K., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: TACAS (2010)
19. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: ESOP (2009)
20. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL (2005)
21. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS (2002)
22. Rosu, G., Ellison, C., Schulte, W.: Matching Logic: An Alternative to Hoare/Floyd Logic. In: AMAST (2010)
23. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: ECOOP 2009 (2009)
24. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: CAV (2008)