# One Step Towards Automatic Inference of Formal Specifications Using Automated VeriFast

Mahmoud Mohsen and Bart Jacobs

iMinds-DistriNet
Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{mahmoud.mohsen, bart.jacobs}@cs.kuleuven.be

**Abstract** VeriFast is a sound modular formal verification tool for C and Java programs. Based on separation logic and using symbolic execution, VeriFast can verify not only memory safety of programs but also full functional correctness. Formal verification is a powerful way of analyzing code, but not yet widely used in practice. Source code has to be annotated with formal specification mostly in the form of function preconditions and postconditions. In this paper, we present Automated VeriFast which is a new extension or an automation layer that lies on top of VeriFast that, given a partially annotated program, offers to attempt to incrementally improve the annotations, e.g. by inferring a fix to the specification of a program fragment that fails to verify. Our thesis is that such small, interactive inference steps will have practical benefits over non-interactive specification inference approaches by allowing the user to guide the inference process and by being simpler and therefore more predictable and diagnosable.

**Keywords:** annotations inference, program verification, separation logic

## 1 Introduction

VeriFast [1], a sound modular verifier for C and Java programs, accepts programs annotated with function preconditions and postconditions written in separation logic [2] and verifies the correctness of the code with respect to these annotations.

Separation logic allows VeriFast to formally prove some properties of programs that were not easy to be proven before, in particular those properties related to pointer manipulations and the heap. However, the process of writing formal annotations makes the verification of real large applications not an easy task. Time and experience in the field of formal methods are required to provide such annotations which become sometimes more complex than writing the source code itself. This motivates the idea of inferring programs' specifications and automating the process of writing the formal annotations. Of course, it is crucial that the user checks that the generated top-level specifications match the program's requirements. Internal specifications need not to be checked.

**The Contributions.** The contribution of this paper is presenting Automated VeriFast which creates an interactive framework in which VeriFast has more automation capabilities that allow users to choose for auto-generating predicates and auto-fixing verification faults detected by VeriFast. The current approach supports some simple linked list patterns, but can be generalized in the future to include more complex data structures, such as doubly linked lists, trees, etc...

## 2 Architecture

Automated VeriFast is a new extension or an automation layer that lies on top of VeriFast verification layer. This new automation layer does not affect in anyway the verification core of VeriFast. This separation ensures that VeriFast's soundness is not affected.

When the user invokes the auto-fix feature after a verification failure, Automated VeriFast takes the verification error message and the symbolic path containing the symbolic states encountered in this path combined with the heap and the stack store in each state as an input. This input is the output of the VeriFast Verification layer.

VeriFast is focused on fast verification, expressive power, and the ability to diagnose errors easily rather than on automation [1]. To accomplish this, VeriFast provides an IDE that facilitates the verification process by allowing users to use the symbolic debugger in the IDE to diagnose verification errors and inspect the symbolic state at each program point.

The user interface of Automated VeriFast is the same except that there are two new buttons in the interface which trigger the new functionalities. One button is for generating the predicates; it should be pressed only once at the beginning of the verification process. The other button is for auto-fixing a verification failure.

Automated VeriFast follows the iterative incremental approach described in [5] to simulate the same manual verification process that normally users of VeriFast follow in solving verification errors. We could put the implementation of the auto-fix within a loop, so one press of auto-fix would either solve all the verification errors at once or stop in a state where an error can't be automatically solved and a manual intervention is required. We did not do that to allow the user to manually intervene at any time in the verification process.

## 3 An Inner Look at Automated VeriFast

In this section, we descibe how Automated VeriFast works in more depth. As mentioned above, there are two new functionalities that have been added to normal VeriFast. The first is automatically generating predicates and the second is auto-fixing errors.

### 3.1 Auto-generating Predicates

A predicate is a named, parameterized assertion [3]. In normal manual verification cases, users define predicates based on their understanding of the different data

structures used within the code. Predicates can be considered as a kind of data abstraction where related data can be encapsulated together in one entity which can be decapsulated later when the data is needed. Moreover, to describe a data structure, such as a linked list or a tree, users of VeriFast have to define recursive predicates which can invoke themselves.

We started in Automated VeriFast by supporting only some simple linked list patterns as a first step. Automated VeriFast generates a predicate for each struct, not only for recursive data structures. Consider you have the following four structs, in Figure 1, that are part of an implementation of a banking system.

```
struct bank {
  int user_account_count;
  struct user_account *user_accounts;
  int bank_account_count;
  struct bank_account *bank_accounts;
};
struct bank_account {
  struct bank_account *next;
  char *id;
  struct user_account *owner;
  int balance;
  int transaction_count;
  struct transaction *transactions;
};
```

```
struct user_account {
  struct user_account *next;
  char *user_name;
  char *password;
  int is_teller;
  char *real_name;
};
struct transaction {
  struct transaction *next;
  char *counterparty_bank_account_id;
  int amount;
  char *comment;
};
```

**Figure 1.** Banking system structs

Automated VeriFast will generate a predicate for each struct that appears in Figure 1. For example, *user_account* is a struct representing a linked list where each node of the list represents one account containing data fields and a pointer to the next account. Automated VeriFast will automatically generate a predicate for the *user_account* struct that looks like:

```
/*@ predicate user_account(struct user_account *user_account; int count) =
  user_account == 0 ? count == 0 :
    user_account->next |-> ?next &*&
    user_account->user_name |-> ?user_name &*& string(user_name) &*&
    user_account->password |-> ?password &*& string(password) &*&
    user_account->is_teller |-> ?is_teller &*&
    user_account->real_name |-> ?real_name &*& string(real_name) &*&
    malloc_block_user_account(user_account) &*&
    user_account(next, ?count1) &*& count == count1 + 1 &*& count > 0;
@*/
```

This predicate takes two parameters. The first is a pointer to the head of the list; the second represents the length of the linked list. The predicate's body has a call to itself which is *user_account(next, ?count1)* that represents the tail of the list encapsulated in a predicate of the same type. The *?count1* is a fresh variable denoting some unknown value representing the length of the list's tail.

The *user_account* predicate is generated without any input from the user, but in some cases, Automated VeriFast requires some hints from users to correctly auto-generate predicates. Users may be required to define some rules of ownership between different structs. As we can see in Figure 1, a struct *bank* instance owns the linked list of struct *user_account* instances pointed to by its field *user_accounts*. It also owns the linked list of *bank_account* instances pointed to by its field *bank_accounts*. Furthermore, it has two counters that represent the length of

both linked lists it owns, namely *user_account_count* and *bank_account_count*. On the other hand, a struct *bank_account* instance doesn't own the linked list of struct *user_account* pointed to by its field *owner* of type *user_account* within its body.

The user can define the ownership relations within the *bank* struct as illustrated in Figure 2 where *owns* and *counts* are new keywords that clarify the ownership relations between structs and allows Automated VeriFast to generate predicates automatically.

```
1 struct bank {
2   /*@ owns @*/ struct user_account *user_accounts;
3   int user_account_count /*@ counts user_accounts @*/;
4   /*@ owns @*/ struct bank_account *bank_accounts;
5   int bank_account_count /*@ counts bank_accounts @*/;
6 };
```

**Figure 2.** Bank struct with ownership annotations

### 3.2 Auto-Fixing

Using VeriFast, both memory safety and full functional correctness of applications can be verified. Therefore, Automated VeriFast was implemented to expect two kinds of errors. The first is memory errors, such as illegal access, buffer overflow, memory leaks, and null pointer dereference; the second is functional correctness errors.

VeriFast supports modular formal verification which means that each function is verified separately and, in case of function calls, VeriFast uses only the callee's contract not its body. To support such modularity, VeriFast implements the frame rule introduced in separation logic which states that while reasoning about a behaviour of a command, it is safe to ignore memory locations not accessed by this command. This allows VeriFast to divide the large heap into small heaplets based on functions' contracts. This concept of locality ensures that any function starts with a heap consisting of what is asserted in the function precondition, which should be taken from the global heap, and at the end of the function the resources asserted by the postcondition will be consumed and returned to the global heap.

From Automated VeriFast's point of view, this facilitates the generalization of all memory errors into two categories: something is needed from the heap, but it doesn't exist in the local heap; something exists in the local heap, but it is not needed. Taking the error message, produced by VeriFast, with other parameters, such as the state of the heap when the error occurred, the stack, the execution context, and the assumptions made out of the specification so far, Automated VeriFast tries to find a solution by generating an annotation. Automated VeriFast can only produce annotations. It doesn't make any changes in the source code. If Automated VeriFast fails to produce any more annotations and the program is not yet successfully verified, then there is either the need for a manual intervention from the user or it may be the case that there is an error in the written code and it can't be verified.

Moreover, Automated VeriFast works on detecting the changes happening to the length of the linked lists in order to infer not only memory specification but also some of the functional properties of linked lists.

## 4 Automated VeriFast by Examples

In this section, we present some examples of using Automated VeriFast to infer formal annotations for some programs manipulating linked lists.

### 4.1 Stack Example

The stack example describes how Automated VeriFast successfully infers formal annotations for some functions that are part of the stack implementation. The Implementation of the stack mentioned in this paper contains two structs. The *node* struct contains a pointer field of its own type *node* that points to the next node and it also contains an *int* value; the stack struct only contains a pointer of type node pointing to the head node. See Figure 3.

Users just need to add the *owns* keyword before the *struct node *head* in the *stack* struct. The predicates shown in Figure 3 will be automatically generated in the source code before the structs.

```
1  struct node
2  {
3      struct node *next;
4      int value;
5  };
6  struct stack
7  {
8      struct node *head;
9  };
```

```
/*@
predicate stack (struct stack *stack; int count) =
    stack->head |-> ?head &*&
    malloc_block_stack(stack) &*&
    node(head, count) &*& count >= 0;

predicate node (struct node *node; int count) =
    node == 0 ? count == 0 :
        node->next |-> ?next &*&
        node->value |-> ?value &*&
        malloc_block_node(node) &*&
        node(next, ?count1) &*&
        count == count1 + 1 &*& count > 0;
@*/
```

**Figure 3.** The node and stack structs are on the left and the auto-generated predicates are on the right

The generated predicates are precise. Precise predicates in VeriFast are similar to precise assertions in separation logic in the sense that for any heap, given a list of input arguments, there is at most one combination of a subheap and a list of output arguments that satisfies the predicate. Precise predicates are declared in VeriFast by writing a semicolon instead of a comma between input parameters and output parameters in the predicate's list of parameter.

Automated VeriFast uses precise predicates mainly for two reasons:

- In general, VeriFast requires the user to insert ghost commands to replace a predicate occurrence by its definition (which is called *opening* the predicate) or vice versa (called *closing* the predicate). If a precise predicate is included in the postcondition, VeriFast tries to automatically open and close it.

– Precise predicates cause VeriFast to infer the predicate's output parameters which helps a lot in proving the functional correctness.

Moving to the use of the auto-fix functionality, verifying the *stack_push* function, appears in Figure 4, using VeriFast, where both precondition and postcondition have empty heaps, raises an error in line no. 7 where VeriFast tries to access the *head* field of the stack while no heap chunk representing this field exists in the heap.

```
1 void stack_push(struct stack *stack, int value)
2     //@ requires true;
3     //@ ensures true;
4 {
5     struct node *n = malloc(sizeof(struct node));
6     if (n == 0) { abort(); }
7     n->next = stack->head;
8     n->value = value;
9     stack->head = n; }
```

**Figure 4.** Push Functions

Automated VeriFast solves the error by adding an annotation that represents the stack and its fields encapsulated in a stack predicate in the precondition of the function. If an error is produced during the verification of a call of this function, then the responsibility will be on the caller not the callee. This preserves the compositional nature of VeriFast and hence Automated VeriFast.

With the new added annotation, VeriFast will next fail to verify the function with a memory leak error and a heap $h$ consisting of node `n` and stack `stack`. To overcome this error, the node fields will be encapsulated into a predicate *node* and then encapsulated with the stack's fields into a stack predicate. Finally this stack predicate will be added to the postcondition of the function. The final contract of the function will be as following:

```
//@ requires true &*& stack(stack, ?count);
//@ ensures true &*& stack(stack, count + 1);
```

The *?count* is a fresh variable denoting some unknown value representing the length of the stack. Automated VeriFast was able to figure out that the length of the stack increased by one as can be seen in the postcondition.

### 4.2 Bank example

The source code of the Bank example is 397 line of code. Automated VeriFast was able to provide annotations for the bank example, excluding the loop invariant annotations, with very few interventions from the user. We will show some examples of these required interventions. Look at the following function:

```
1 void socket_write_transactions_helper2(struct socket *socket, int count, struct transaction
      *transactions)
2 //@ requires true &*& transaction(transactions,?count1) &*& count1 > 0 &*& socket(socket);
3 //@ ensures true &*& transaction(transactions,count) &*& socket(socket);
4 {
5     ..........
6     ..........
7 }
```

Automated VeriFast was able to generate the shown pre/post-condition. The user needs to change the *count1 > 0* condition in line 2 in the precondition and writes instead *count1 == count* and the function will be verified successfully.

Another example is the following function where the post-condition, auto-generated by Automated VeriFast, has to be slightly modified by the user to be successfully verified:

```
1  struct authenticate_result *authenticate_user(struct user_account *userAccounts, char *userName, char
       *password)
2  //@ requires true &*& user_account(userAccounts,?count0) &*& string1(userName) &*& string1(password);
3  //@ ensures true &*& string1(userName) &*& string1(password) &*& user_account(userAccounts,count0)
       &*& authenticate_result(result);
4  {
5     if (userAccounts == 0) {
6              return 0;
7        }
8      else {
9        ..........
10       ..........
11    }
12
13 }
```

To successfully verify this function, the user has to put the last part of the post-condition *authenticate_result(result)* within a conditional statement like the following: *(result == 0 ? true : authenticate_result(result))*. Other than these kinds of possible interventions everything else is almost auto-generated except for the loop invariant which still needs to be manually provided.

## 5 Related Work

Infer [4] is a static analysis tool based on separation logic. It performs a deep heap shape analysis. Infer is able to automatically generate pre/post-conditions, but it focuses on detecting only memory errors, ignoring still a wide range of other possible functional errors. Our approach is different than the one followed by Infer. We don't claim to compete with it as we don't have a static analysis tool, but rather an automated verification tool. Static analysis tools' main goal is to find bugs, but our goal is to verify code.

Some work that was already done with VeriFast to gain some automation capabilities can be found in [5]. Automated VeriFast uses from this work the auto-open and the auto-close, but it has more functionalities, such as auto-generating predicates and inferring both the precondition and the postcondition.

Another work that shares the same aim which is inferring annotations automatically for VeriFast, but with a different way of approaching that is presented in [6]. They are using machine learning and dynamic analysis to capture some behaviours of programs that allow them to automatically generate annotations and feed it to VeriFast. Using dynamic analysis may end up giving good results regarding the shape of data structures, but it is still a headache for users to generate test suites whose quality will definitely affect the results of the dynamic analysis.

## 6 Conclusions and Future Work

The goal of our work is not to completely eliminate the need for user effort, but to reduce the annotation effort required as much as possible. In this paper, we presented Automated VeriFast which creates a framework in which the user can use the auto-generate predicates and the auto-fix functionality to solve verification errors, choose to write his own annotations manually, or combine both automation with his experience in writing formal annotations.

The current approach of Automated VeriFast supports some simple linked list patterns in which the linked list can be manipulated by adding or removing nodes to and from the list. We are working now on extending it to support additional patterns. We will work also on inferring the loop invariants. Furthermore, we will focus on inferring more specifications that prove functional correctness for more complex applications. The source code of Automated VeriFast is available at [9].

**Acknowledgments.** This work was funded by the Flemish Research Fund through grant G.0058.13.

## References

1. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: APLAS (2010)
2. O'Hearn, P.W: A primer on separation logic (and automatic program verification and analysis). Software Safety and Security; Tools for Analysis and Verification. NATO Science for Peace and Security Series 33 (2012), 286–318.
3. VeriFast tutorial, http://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf
4. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NFM (2011)
5. Vogels, F., Jacobs, B., Piessens, F., Smans, J.: Annotation Inference for Separation Logic Based Verifiers. In: FMOODS/FORTE (2011)
6. Muhlberg, J. T., White, D. H., Dodds, M., Luttgen, G., and Piessens, F.: Learning Assertions to Verify Linked-List Programs. In: SEFM (2015)
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2005)
8. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
9. https://github.com/Mahmohsen/verifast/tree/Automated-Verifast
10. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Pur- brick, J., Rodriguez, D.: Moving fast with software verification. In: NFM (2015)