Automated VeriFast: Supporting Formal Specification Authoring through Specification Inference

Mahmoud Mohsen
mahmoud.mohsen@cs.kuleuven.be

Bart Jacobs
bart.jacobs@cs.kuleuven.be

iMinds-DistriNet, Dept. C.S., KU Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium

ABSTRACT

VeriFast is a sound modular formal Verification tool for C and Java programs. It accepts programs annotated with preconditions and postconditions written in separation logic as an input and verifies the correctness of the code with respect to these annotations. 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. Our current prototype is able to infer preconditions, postconditions, and loop invariants for some programs that manipulate linked lists.

Keywords

Automated VeriFast; Annotation Inference; Separation Logic

1. INTRODUCTION

Authors in [4] mentioned that they focused more while implementing VeriFast on fast verification, expressive power, and the ability to diagnose errors easily rather than on automation. This is what differentiates VeriFast from other tools, such as Smallfoot or jStar [1, 3] which focus on automatic verification. Infer [2] reflects the latest development of the latter tools adopting the approach of full automation. This approach facilitates the integration of Infer, as a static analysis tool, into the software development cycle at Facebook. From another perspective, Infer had to accept less preciseness in favour of being fully automated. These tools are handling memory correctness, but they ignore a wide range of the other possible functional errors.

Automated VeriFast, on the other hand, follows another approach, similar to the iterative incremental approach described in [5], in which the final goal may not be a full automatic verifier, but certainly a verifier that requires much less effort and time from users. Using VeriFast, programs can be verified for memory and functional correctness. Therefore, Automated VeriFast keeps track of the changes happening to the length of the lists in order to infer not only memory specification but also some of the functional properties of linked lists.

2. SYMBOLIC EXECUTION

VeriFast is a modular verification tool in the sense that it symbolically executes each routine separately and refers to other routines' contracts to verify calls. The symbolic execution step can be described by symbolic transition relations, which are relations from initial symbolic state σ to outcomes o. A symbolic state $\sigma = (\sum, h, s)$ consists of $\sum$ representing assumptions, written in first order logic, generated based on annotations that have been already verified leading to this state, a symbolic heap h which contains permissions, known as chunks in VeriFast, for accessing certain memory locations, and a symbolic store s. The output o is either a final symbolic state or an error **abort**.

Automated VeriFast is an automation layer above the VeriFast core layer. The source code is first verified by VeriFast; if the result is **abort**, Automated VeriFast will add more annotations trying to find a symbolic transition that can transfer the last sybmbolic state, before **abort**, to another state $\sigma' = (\sum', h', s')$ where $\sigma' \neq$ **abort**. If Automated VeriFast is unable to add any more annotations and $\sigma' =$ **abort** then the user has to intervene because either a problem exists in the source code or there is a limitation in Automated VeriFast.

3. THE AUTOMATION LAYER

The automation layer, built on top of VeriFast, introduces two new functionalities triggered by two new buttons added to the VeriFast interface, namely Auto-Predicate and Auto-Fix. The first button should be pressed once at the beginning of the verification process to automatically generate predicates for different data structures used within the program. As a general rule, Automated VeriFast generates predicates for all C structs defined in the program.

The second button is for fixing errors detected by the VeriFast core layer. These two new functionalities create an interactive framework in which the user can either accept the solution provided by Automated VeriFast, choose to write his own annotations manually, or combine both automation with his/her experience in writing formal annotations.

4. ABSTRACTION AND RECURSIVE PREDICATES

Predicates, in VeriFast, are a data abstraction technique in which related data can be encapsulated together in one entity that can be decapsulated later when the data is needed. User defined predicates are required to describe an inductive data structure, such as a linked list or a tree. Users

can define recursive predicates which invoke themselves to describe such inductive data structures.

For example, the following struct represents a list of nodes, where each node has a data field and a pointer to the next node:

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

Automated VeriFast auto-generates the following recursive predicate for the node struct:

```
/*@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; @*/
```

These recursive predicates are essential especially to generate the loop invariant; otherwise, the size of the heap can become unbounded if new heap allocations happen within the loop body. Folding/unfolding predicates or as we call them in this paper encapsulating and decapsulating predicates are the way to gather nodes of the same linked list together in one predicate or separate them in different nodes. In VeriFast, this can be accomplished using the commands *open* and *close*.

Although VeriFast has the functionality of *auto-open* and *auto-close* presented in [5], Automated VeriFast has, in some cases, to auto-generate the *open* or *close* command on its own.

5. INFERRING ROUTINES' CONTRACTS

Routines' contracts are written in the form of **requires** and **ensures**. The keyword **requires** consumes the heap chunks that are required by the routine from the global heap during the verification of the routine call and produces the heap chunks to the local heap during the verification of the routine itself. On the contrary, **ensures** produces the heap chunks to the global heap during the routine's call verification and consumes the heap chunks from the local heap of the routine after verifying the routine's body. This implementation of **requires** and **ensures** is based on the *frame rule* in separation logic which introduced the concept of locality.

This concept of locality allowed Automated VeriFast to limit its expectation of memory errors into two categories. Either there is a need for a heap chunk that doesn't exist in the heap or there is a heap chunk that exists and it is not needed.

The following example is a *stack_push* routine, which is part of a *stack* implementation, in which a new node is pushed into the stack:

```
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; }
```

Verifying *stack_push* using VeriFast in this form, 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. 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 routine. 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 property of VeriFast and hence Automated VeriFast.

With the new added annotation, VeriFast will next **abort** 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 it can be seen in the postcondition.

6. LOOP INVARIANT

Automated VeriFast deals with loops as if it deals with recursive routines. The loop invariant is written in the form of pre/postcondition. A loop is considered by Automated VeriFast as a routine that contains a call to itself and keeps calling itself recursively as long as the loop's condition holds. When the loop's condition is true and the loop enters one or more iterations, the heap chunks required by the loop's precondition should exist in the local heap of the loop before the start of each iteration.

Acknowledgments.

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

7. REFERENCES

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO November 2005, Revised Lectures*, pages 115–137, 2005.

[2] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NFM April 2011. Proceedings*, pages 459–465, 2011.

[3] D. Distefano and M. J. Parkinson. jStar: towards practical verification for java. In *OOPSLA October 2008*, pages 213–226, 2008.

[4] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS November 2010. Proceedings*, pages 304–311, 2010.

[5] F. Vogels, B. Jacobs, F. Piessens, and J. Smans. Annotation inference for separation logic based verifiers. In *FMOODS, FORTE June 2011. Proceedings*, pages 319–333, 2011.