Dafny verification on simply memory management

Chongxin Luo(cluo5), Vignesh Sridhar(vsridha2)

March 2017

Contents

1 Introduction

Program verification is a complicated thing to learn let along apply into modern software engineering but one that could solidify the discipline. Program verification is a cornerstone of program analysis, and competency in it could lead to improvement in program synthesis. [2] Likening software engineering to language, analysis is reading and synthesis is writing, we must be able to read well to be able to write well. They go hand in hand and allow for improvement to be made across the system, and could lead to development globally and establish the discipline.

2 Motivations

Software testing is currently the most employed method to find error in code, but does not show their absence. The absences of bugs that come about during software testing indicate either high quality program synthesis or poor testing, and to distinguish between requires additional information. It is quite hard to be able to test all potential inputs and outputs during software testing which will show the correctness of the program. Formal verification methodology employing a variety of techniques which generate and evaluate possible cases allows for proving correctness.

Having code which has been formally proved to be correct provides many advantages. Security issues are usually cased by malicious taking advantage of unintended handling of edge cases or overflowing certain areas. Verified programs will not run into these issues, in our case we are building a sample of memory management. Issues here could lead to data loss or lack of operation, verified software will have the confidence that these issues won't arise.

Our project is a small foray in to this universe, we hope to garner some insights which will allow for us to become better software engineers. Dafny is a powerful verification language developed by Microsoft which leverages the power of modern Satisfiability modulo theories(SMT) solving algorithm to raise flags to developers [3]. The Dafny verifier runs in the compiler and as such a programmer interacts with it like a static type checker, typing to solve the errors raised by this tool. We will be using Dafny to verify a memory management program to understand the capabilities of Dafny as we ll as understand how we can write programs with better logic. Verification of this scale comes from understanding the logic flow of the code and representing it using arithmetic logic expression.

3 Top level implementations

Our software implements some of the basic functions needed for memory management and to demonstrate the correctness of our implementations. We cover allocation, deallocation, the bookkeeping needed to keep track, and some test functions which make different memory calls. For our memory, we have decided to represent them as an array that indexes two hundred elements, if a memory location is being used we place the process number in it, zero otherwise. This methodology was done as our goal is to test the way the memory is managed and not the I/O capabilities. Also we decided on two hundred elements as it is a size-able amount that will allow us t o explore plenty of processes. The array will contain a process number which indicates a given process taking up that memory location which can be symbolic of a chunk of memory being taken up by a process. The allocation and deallocation function units works under the bookkeeping control module, all three components work together to achieve the functionality of a simply memory management program. The top level memory management module flow chart is shown in the figure below.

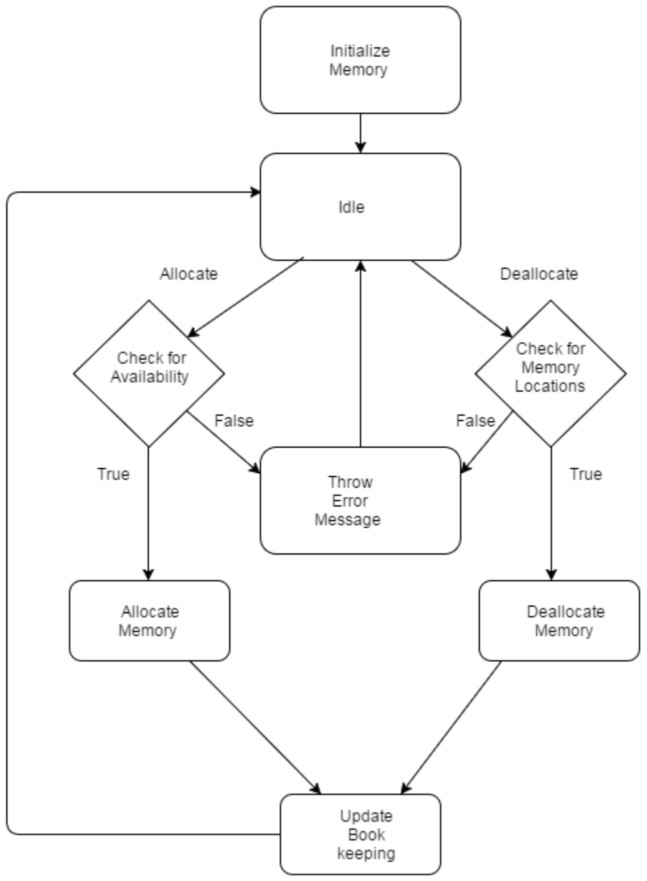

Figure 1: memory management top level flow chart

3.1 Memory Allocation

The first functional components is the the memory allocate module. The memory allocate function which takes in a process number, the size of memory required for the process, the memory array, and the bookkeeping map. The output of this is a boolean value which indicates success or failure and also the updated bookkeeping map. The memory allocate function requires the function input to be non-empty, which means the input memory array and bookkeeping map cannot be null and should have size greater than zero. Most importantly, the memory allocate function requires the input memory array and bookkeeping map to be successfully linked. This means every single occupied memory locations inside the memory array is being correctly tracked and stored in the bookkeeping map, and every entries in the bookkeeping map has a correct corresponding locations and process number in the memory array. We call this link between memory array and bookkeeping map as goodMemoryLink, this is the most important logic check for the entire memory management program. The goodMemoryLink contains a set of conditional logic in order to check the valid state of the memory array and bookkeeping map, and it is being used widely in the entire memory management program. In order to reduce our code complexity and achieve a better dafny code structure, we used dafny function to implement the goodMemoryLink conditional logic check.

```
function goodMemoryLink(mem: array<int>, table:map<int, int>):bool
requires mem != null && mem.Length > 0;
reads mem;
{
  forall m:int :: 0 <= m < mem.Length ==> mem[m] != 0 ==> m in table
 && forall m:int :: 0 <= m < mem.Length ==> mem[m] == 0 ==> m !in table
```

```
  && forall m:int:: m in table ==> 0 <= m < mem.Length
  && forall m:int:: m in table ==> mem[m] == table[m]
}
```

A dafny function specification is zero or more reads, requires, ensures or decreases clauses, in any order. A function specification does not have modifies clauses because functiosn are not allowed to modify any memory. [3] We used the dafny function module, takes input of the memory array and bookkeeping map, going through the set of conditional logic, and returns a boolean variable as the output. This specific functionality of danfy function allowed us to use it as a advanced implementation for the pre-condition and post-condition.

With the correct pre-conditions satisfied, the memory allocate function initially checks the number of empty slots inside the memory array, and compare it with the memory size required by the process. If the empty memory slots are less than the required memory size, the memory allocate function stops and return false. Otherwise, it loops through the entire entire memory array, find the first opening spot in the memory array, and allocate it for the process number, update the bookkeeping map and iterate till we have met the required memory size. If the memory is being successfully allocated, the allocate function returns true, and returns the new updated bookkeeping map. At the end of the allocate function, it need to ensure the goodMemoryLink still holds, which indicate the memory modification did not mess up the memory management tracking structures, and it will be used as pre-condition for following function calls.

3.2 Memory Deallocation

The deallocate function works a lot similar as the allocate function. It takes the input of process number, memory array and bookkeeping map. The function requires both memory array and bookkeeping map to be not null, and obviously, it requires the goodMemoryLink condition to be true between the given memory array and bookkeeping map. When the deallocate function is being called, it starts iterating through the bookkeeping map and check every process number and if it matches with the given process number, it removes the proces number from the memory cell and set the corresponding memory location in the memory array to zero. In the end, it delete the corresponding process in the bookkeeping map, and ensures the goodMemoryLink holds in the end of the function call. The deallocate function returns a boolean variable as an updated bookkeeping map, if the process is being successfully deallocated from the memory, it returns true, and the newly updated bookkeeping map. Otherwise, it returns false, and returns the old bookkeeping map.

3.3 Memory Management

The memory management function works as the top level of the program, it initially allocate both memory array and bookkeeping map, and perform the memory link between them to establish the initial goodMemoryLink in the program. The three initial functions is shown below.

```
/*memory initialization*/
method initMemory(size: int) returns (memory: array<int>)
requires size > 0
ensures memory != null
ensures memory.Length == size
ensures forall m:: 0 <= m < memory.Length ==> memory[m] == 0
ensures fresh(memory)
{
        memory := new int[size];
        forall(i | 0 <= i < memory.Length){
                memory[i] := 0;
        }
        return memory;
}
```

```
/*bookkeeping table initialization*/
method initTable()returns(table: map<int,int>)
ensures |table| == 0;
{
  table := map[];
}

/*memory link initialization*/
method linkMemory(mem: array<int>, table: map<int,int>)
  requires mem != null;
  requires mem.Length > 0;
  requires forall m:int::0 <= m < mem.Length ==> mem[m] == 0;
  requires |table| == 0;

  ensures goodMemoryLink(mem, table);
{

}
```

With the initialization complete, the Memory Deallcoation function waits for a system calls, which will give it a process number, required process memory size, and allocate/deallocate action. With the input parameters, it calls either the allocate function or deallocate function, and checks the output states. If the output is false, it print out the error message and reject the specific memory modification actions. If the output is true, it takes the output bookkeeping map and update the original bookkeeping map with the new one. Because the Memory Management function is on the most top level of our memory program, it requires no pre-conditions and post-conditions.

3.4 Helper functions

In order to support the memory program structure and program verification, several helper functions are being implemented with specific purposes. One of the helper functions that we implemented with this purpose is RecursiveCount function. The RecursiveCount function uses the recursive method to count the number of empty spots inside the memory. The function takes input the memory array, a key that being counted, and the index location that in the array that needed to be counted from. The function recursively calls itself with index keep increasing until it reaches the end of the memory array, and returns the sum of number of keys in the array. The function is being called with key equals to zero, which indicates the empty spot in the memory array, and it returns the number of empty spots in the memory array. The code for RecursiveCount function is shown below.

```
 function RecursiveCount(key: int, a: array<int>, from: nat) : int
    reads a
    requires a != null
    requires from <= a.Length
    decreases a.Length-from
{
    if from == a.Length then 0
    else if a[from] == key then 1+RecursiveCount(key, a, from+1)
    else RecursiveCount(key, a, from+1)
}
```

4 Implementation and Verification Challenges

During our time developing this software, we went through multiple iterations and variations due to constraints we came across or following a rabbit hole that kept going deeper and deeper. In our first approach we used a 2 dimensional array for bookkeeping purposes, it was a 200x200 array. Each index corresponding to a process number and could contain up to 200 potential memory locations. The limitation is that our process number could only be from 0 - 199 and also it's a very inefficient data structure. It is quite hard to keep track of used indices and free indicies.

In order to fix the above problems, we moved on to using maps, individual memory locations as the key and corresponding process number as the value. This structure gives us a lot more freedom in the implementation, the process number is not constrained between 0-199 anymore, instead, the process number can be any integer value, and the memory locations are also flexible as well, where the same bookkeeping structure can be used for different memory array size. However, the problem of this implementation is it inefficient on the memory usage and computational time. For each process number, it has multiple entries in the map which corresponding different locations in the memory array. This means that for deallocate, we need to loop through the entire map to find all the corresponding memory locations to perform deallocation on.

To move away from both implementations, we tried the third implementation which uses process number as the key and array of memory locations as the value. This structure was the most efficient approach, and also had no constraints on both process number of memory array size. The figure below visually shows the memory array and bookkeeping implementation of using the map with arrays.

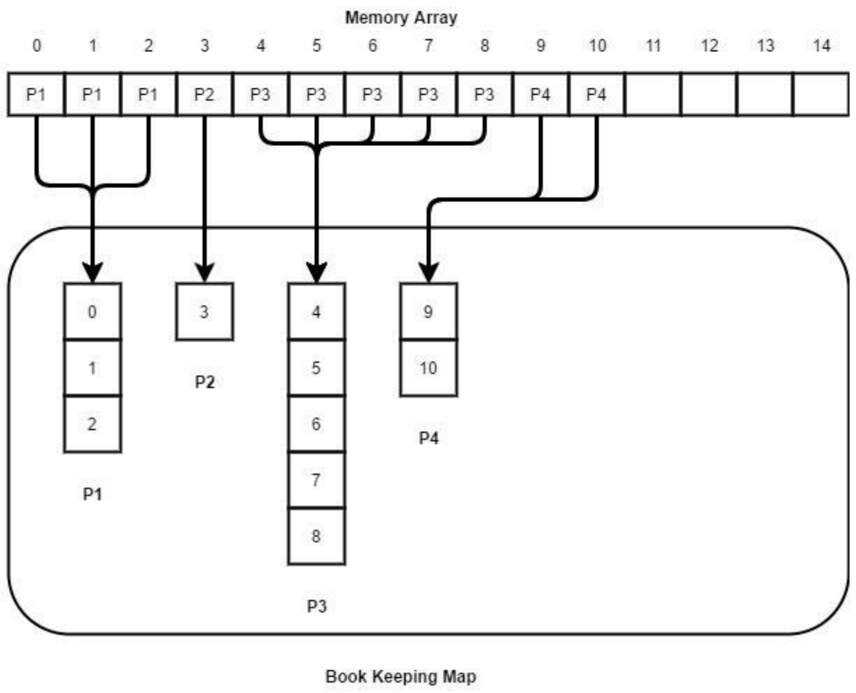

Figure 2: Bookkeeping Map and Memory Array

On the side of verification, dafny has a whole array of tools and specifications that allow us to verify our program. We used functions, methods, predicates, collection types, etc... Function specification is zero or more reads requires ensures or decreases clauses. A function specification does not have modifies clauses because functions are not allowed to modify any memory. [1] We use the function type to construct the memory linage pre- and post- conditions. Method specification is zero or more modifies, requires ensures, or decreases clauses. [1] Methods are different from function such that it is allowed to modify given memory positions, which is a powerful actions we can use in our memory management program. Predicate is a function that returns a boolean result.

A linkage function is used to connect the bookkeeping map to the memory array as discussed above, this is a check that to occur before and after any function call that clobbers either the memory array or the bookkeeping map. It replaces the role of precondition and post condition. The linkage is an if and only if condition. It checks the bookkeeping map against the memory array and vice versa. Initially we didn't have any linkage function instead we had very large and complex conditional logic statements as pre-conditions and post-conditions. The problem we were running into was that it became unruly and difficult to keep track of, and it is hard to keep it consistant. So we created the dafny function that specifically takes in memory array and bookkeeping map, runs all the conditions and returns either success or failure.
With the linkage function put into place, we ran into some difficulties to keep the condition satisifies during the program. More specifically, we are not able to find a way to indicate number of empty spots inside the memory array as a pre-condition for the allocate function, it is hard to maintain the linkage function true throughout the allocation function. The same issue occurred when we are verifying the deallocation function.

We initially wrote our memory management program in C++, and tested many corner cases in C++, and after that, we transfered the C++ code into dafny and started dafny verification on the memory program. In order to test and verify our program, we performed two types of tests, individual functionality test and a test of the integrated system. For individual test, each function is separately run through dafny with given pre-conditions and post-conditions. This test allows us to determine the correctness of each function, and it helps to divide a big program into small sections, and verify each individual sections are easier. This was quite useful initially to be able to debug many of the functions and also give us an understanding of the limitation and the capabilities of each function. With the individual functionality test completed, we moved on to testing with the entire integrated system. During this test, we simulated different cases with different allocation requests and deallocation requests, to make sure we tested all possible actions that might be called on the memory module. The integrated system test runs mimicked a computer operating system interacting with the memory system. Of course ours is a very scaled down version of an actual memory system, but the principle are applicable through out and can be applied for a full memory system.

5 Conclusion

Unfortunately, we were unable to completely verify our program, we were able to pass the modular verifications but the integrated software was unable to pass conditions of the linkage. This is due to the fact that we were unable to count the number of empty spots in the memory array as a precondition. We were able to identify this as the weak spot in our program as we hard coded in an empty memory array before every allocate function and we were able to pass through.
We were unsure if this is due to our misunderstanding of dafny constraints that doesn't allow us to acquire the number of empty spots or our overall approach to this code in dafny that led us to that issue. We were able to detect what the issue was and there were a few alternative to this situation. We could weaken the verification system and make it so that there is always enough open spots. This would lead to a memory management system with holes in it leading to data loss, system failure, and a whole host of other problem.

We believe that a huge portion of the difficulties we faced was due to lack of documentation for the syntax and also the small community. Not having references to look at when writing programs like this led us down various paths which turned out to be dead ends, delaying us in getting to our end goal.

Verification can demonstrate the soundness of any piece of code and ensure users reliability and security. It is a powerful tool which will hopefully be adopted universally.This is the power of Dafny, with every sub block having its own pre- and post- conditions, along with the logic expressions enabling us to verify in a modular fashion and piece it together. This scalability is also another benefit of Dafny as it can be quickly adopted by enterprises to secure their own software. It is also something developers should have proficiency in as it will help strengthen their coding ability. We learned from this experience to break up big chunks of software and verify them in a modular fashion. In terms of coding, this experience makes us think beyond corner cases, its about the overall logical flow and also whether we are truly solving the problem.

References

[1] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness.

[2] Ali Mili. A case for teaching program verification:its importance in the cs curriculum. 1983.

[3] K. Rustan M. Leino Richard L. Ford. Dafny reference manual, 2016.