

Verification Experimentation with Implicit and Explicit Style Specifications

Beth Anne Byrd and Heather Harton

Abstract—A major goal of a verification condition (VC) generation system is to generate assertions that are easily provable. Previous research has included categorizing VCs to assist in understanding the difficulty of the resulting proof. An open question is whether a VC can move from a category that requires a more difficult proof into a category where the proof can be automated by using explicit functional specifications. This paper summarizes results from our experimentation with alternative specifications of a List component.

Index Terms—Verification, Specification, Implicit, Explicit

I. INTRODUCTION

GENERATING verification conditions for code can result in varying degrees of ease of provability. Manually proving the conditions can help determine where the verifier needs to be improved and assist in the design of an automated prover.

The method of specification has a significant impact on the ease of verification. While relational specifications are necessarily implicit, functional specifications may be presented in an implicit and explicit style. We hypothesize that when functional specifications are given in an explicit style, the resulting verification conditions (VCs) will be easier to prove. This is because it is possible to use a simpler proof rule for handling operation calls with explicit specifications [2], leading to more directly provable VCs. Towards this end, we are modifying some current specifications into an explicit functional style. For example, consider a stack which is modeled by two mathematical strings. Previously, the ensures clause of the Pop operation was specified as: $\#S = \langle R \rangle \circ S$ where the outputs of the operation (R and S) when concatenated are equal to the starting value of the variable S in Pop. However, we could modify the specification to use a `Prt_Btwn` function which provides a mechanism to write the specification of Pop in an explicit manner so that the output of an operation is written in terms of the inputs. There is a cost to the use of the `Prt_Btwn` definition and it goes beyond developing a simple definition; it also involves development of

suitable theorems for automated proving.

The `Prt_Btwn` function subdivides strings. It takes three parameters: the index where the resulting string should begin, the index where it should end, and the string it is subdividing. For instance, if a string $S = \langle 1, 2, 3, 4, 5 \rangle$ and `Prt_Btwn` is called with: `Prt_Btwn(1, |S|, S)`, the resulting string is $\langle 2, 3, 4, 5 \rangle$. `Prt_Btwn` returns the string indexed from 1 up to and including the “to” index. The `Destring` operator takes a string containing a single entry and returns that entry. So, the ensures clause of the Pop operation using `Prt_Btwn` would be:

Operation Pop (updates S: Stack; replaces R: Entry);
requires ...
ensures $R = \text{Destring}(\text{Prt_Btwn}(0, 1, \#S))$
 and $S = \text{Prt_Btwn}(1, \#S, \#S)$;

II. EXAMPLES

This paper builds on previous work on categorizing VCs in [1], that describes four categories of generated VCs and analyzes a library of generated VCs from Ohio State Verification Condition Generator. This section will describe some generated VCs from the VC Generator developed at Clemson University which includes a minimalist, rewriting prover [3].

The majority of the examples that we will examine use one of two versions of a List Template. An abbreviated specification of both versions is given in the Appendix; the first version has operations with implicit specifications while the second has operations with explicit specifications. Both versions use a two-string mathematical model of lists. The first string holds all of the entries in the list in front of the current list position; the second holds the remaining entries, beginning with the entry in the current list position. The following specification and implementation of the List Reversal operation are used in the example VCs in this paper.

Operation Reverse (updates S: List_Position);
requires $|S.Prec| = 0$;
ensures $S.Prec = \text{reverse}(\#S.Rem)$ and $|S.Rem| = 0$;

This research is funded in part by National Science Foundation grants CCF-0811748 and DMS-0701187.

Heather Harton is a Ph.D. student at Clemson University. (e-mail: hkeown@clemson.edu).

Beth Anne Byrd is a student at Clemson University (e-mail: ebyrd@clemson.edu).

```

Procedure Reverse(updates S:List_Position);
  decreasing |S.Rem|;
  Var temp: Entry;
  if Rem_Length(S) > 0 then
    Remove(temp, S);
    Reverse(S);
    Insert(temp, S);
    Advance(S);
  end;
end Reverse;

```

A. VCs Provable Without Assumptions

The most basic verification conditions are those which are easily provable without even needing to refer to the given assumptions. The original paper described this category as verification conditions in which the goal is provable using only rules of mathematical logic. However, we have altered this definition, as this category only contains VCs which are so obvious that the assumptions are rendered unnecessary. The following example uses the explicit specification.

```

(min_int <= 0) and
(0 < max_int) and
(|S.Prec| = 0) and
(P_val = |S.Rem|) and
(|S.Rem| > 0) and
(<??temp> = Prt_Btwn(0, 1, S.Rem)) and
(???S.Prec = S.Prec) and
(???S.Rem = Prt_Btwn(1, |S.Rem|, S.Rem))
=====>
(|Prt_Btwn(1, |S.Rem|, S.Rem)| < |S.Rem|)

```

Example Verification Condition from List Reversal

Without having to refer to any of the assumptions given, we can deduce that since $\text{Prt_Btwn}(1, |S.\text{Rem}|, S.\text{Rem})$ is equal to $S.\text{Rem}$ excluding the first element, so it is apparent that $|\text{Prt_Btwn}(1, |S.\text{Rem}|, S.\text{Rem})| < |S.\text{Rem}|$ is true. No further proof is necessary.

The corresponding VC generated using the implicit specification, however, falls into category C or D and requires a more complex proof. This seems to support our hypothesis that explicit specification may increase the ease of verification.

B. VCs Provable Using Assumptions

Occasionally, a generated verification condition is exactly the same as an assumption. The verifier is a work in progress, so this category will eventually be eliminated as improvements are made. An example of this category comes from a list reversal operation; the specification and implementation follow. An exact duplication of the goal is an example of the category which includes any VC which requires some number of hypotheses to prove the VC.

```

(min_int <= 0) and
(0 < max_int) and
(|S.Prec| = 0) and
(P_val = |S.Rem|) and
(|S.Rem| > 0) and
(<??temp> = Prt_Btwn(0, 1, S.Rem)) and
(???S.Prec = S.Prec) and
(???S.Rem = Prt_Btwn(1, |S.Rem|, S.Rem))
=====>
|S.Prec| = 0

```

Example Verification Condition for List Reversal

This is clearly true, because one of the given assumptions is that $|S.\text{Prec}| = 0$. So, this condition is also very easily provable. The related VC for the implicit specification also belongs in this category.

C. VCs that Require General Mathematical Theorems for Proof

Another category of provable verification conditions includes those that are easily proven with only some simple substitutions and simplifications based on the information given in the assumptions. This information generally is available in the included mathematical theories and facts from the assumptions. This will vary from prover to prover, however, as different provers may have various degrees of definitions built in. From the list reversal example (explicit specification):

```

(min_int <= 0)and
(0 < max_int) and
(|S.Prec| = 0) and
(P_val = |S.Rem|) and
(|S.Rem| > 0) and
(<??temp> = Prt_Btwn(0, 1, S.Rem)) and
(???S.Prec = S.Prec) and
(???S.Rem = Prt_Btwn(1, |S.Rem|, S.Rem)) and
(???S.Prec = reverse(Prt_Btwn(1, |S.Rem|, S.Rem))) and
(|???S.Rem| = 0) and
(Entry.is_initial(?temp)) and
(??S.Prec = reverse(Prt_Btwn(1, |S.Rem|, S.Rem))) and
(??S.Rem = (<??temp> o ???S.Rem)) and
(?S.Prec = reverse(Prt_Btwn(1, |S.Rem|, S.Rem)) o
Prt_Btwn(0, 1, (<??temp> o ???S.Rem))) and
(?S.Rem = Prt_Btwn(1, |(<??temp> o ???S.Rem)|, (<??temp>
o ???S.Rem))
=====>
reverse(Prt_Btwn(1, |S.Rem|, S.Rem)) o Prt_Btwn(0, 1,
(<??temp> o ???S.Rem)) = reverse(S.Rem)

```

Example Verification Condition for List Reversal

Given that $\text{Prt_Btwn}(0, 1, S.\text{Rem})$ and $|\text{???S.Rem}| = 0$ which means that $\text{???S.Rem} = \text{empty_string}$,

we can see that $\text{Prt_Btwn}(0,1, \langle \text{temp} \rangle \circ \text{S.Rem})$ from the left hand side will simplify to $\text{Prt_Btwn}(0, 1, \text{S.Rem})$.

So, the left hand side is $\text{reverse}(\text{Prt_Btwn}(1, |\text{S.Rem}|, \text{S.Rem})) \circ \text{Prt_Btwn}(0, 1, \text{S.Rem})$, which is equivalent to $\text{reverse}(\text{S.Rem})$.

So, this VC is also proven to be true.

D. VCs That Are Provable with Programmer Supplied Definitions

Proving some verification conditions can sometimes call for other knowledge outside of the information given in the assumptions, beyond basic mathematical theories. For instance, as the following example demonstrates, one might need to know that the length of a string can never be negative. For some provers, the distinction between categories C and D makes no difference because there are not many definitions built in to the prover. The example for this category also comes from the list reversal implementation using the explicit specification.

```
(min_int <= 0) and
(0 < max_int) and
(|S.Prec| = 0) and
(P_val = |S.Rem|) and
not(|S.Rem| > 0)
=====>
S.Prec = reverse(S.Rem)
```

Based solely on the assumptions, this verification condition is not provable; all we know is that the length of S.Prec is zero, not how S.Prec is related to S.Rem . Therefore, some knowledge about the nature of strings is necessary in order to prove that this statement is indeed true.

We know that the length of any string can never be negative. In fact, the length of any string must always be positive unless it is equal to the empty string, which means its length is zero.

The given assumption $\text{not}(|\text{S.Rem}| > 0)$ implies that $|\text{S.Rem}| \leq 0$. Since we know that the length of a string cannot be negative, this in turn implies that $|\text{S.Rem}| = 0$, so S.Rem is the empty string. Since S.Prec and S.Rem are both equal to the empty string, we can conclude that $\text{S.Prec} = \text{reverse}(\text{S.Rem})$.

For the corresponding VC using the implicit specification, there is no change in the categorization.

E. Currently Unprovable VCs

Sometimes verification conditions are generated that are currently unprovable. This can either mean that there is a

problem with the VC generation, or with a specification, annotation, or implementation, or with the library of theorems and current techniques for automated proving.

III. LIST REVERSAL

Verification conditions for this paper were generated for the List Reversal example using two different List Template specifications. Each generated eight VCs. A summary table of how many VCs were generated in each category can be found in the Appendix.

For the original List Template specification, most of the VCs are in category D: provable with programmer supplied definitions. VCs 0_2 , 0_4 , 0_6 , 1_1 , and 1_2 all fall into this category. This is natural because the specification is string-based, so some knowledge of the nature of strings is then necessary to prove the VCs. One VC (0_3) was provable using assumptions, and was used earlier as an example. The remaining two VCs (0_1 and 0_5) were provable using general mathematical theories and thus belong in category C.

The VCs for explicitly functional List Template specification are similarly grouped. Most are provable with programmer supplied definitions (category D): 0_4 , 0_6 , 1_1 , and 1_2 . VC 0_2 was previously shown as an example for category A, those provable even without the assumptions. One VC, 0_3 , belongs in category B. General mathematical theories are sufficient to prove the rest of the VCs, 0_1 and 0_5 (category C).

Two corresponding VCs from the two sets are below:

```
(0 < max_int) and
(min_int <= 0) and
(|S.Prec| = 0) and
(P_val = |S.Rem|) and
(|S.Rem| > 0) and
(????S.Prec = S.Prec) and
S.Rem = (<temp> o ???S.Rem) and
(???S.Prec = reverse(???S.Rem) and
(|???S.Rem| = 0) and
(??S.Prec = reverse(???S.Rem)) and
(??S.Rem = (<temp> o ???S.Rem)) and
((?S.Prec o ?S.Rem) = (reverse(???S.Rem) o (<temp> o
???S.Rem))) and
(|?S.Prec| = |reverse(???S.Rem)| + 1)
=====>
?S.Prec = reverse(S.Rem)
```

VC 0_5 Generated from Implicit Functional List Specification

```

(min_int <= 0) and
(0 < max_int) and
(|S.Prec| = 0) and
(P_val = |S.Rem|) and
(|S.Rem| > 0) and
(<??temp> = Prt_Btwn(0, 1, S.Rem) and
(???S.Prec = S.Prec) and
(???S.Rem = Prt_Btwn(1, |S.Rem|, S.Rem)) and
(???S.Prec = reverse(Prt_Btwn(1, |S.Rem|, S.Rem))) and
(???S.Rem = 0) and
(Entry.is_initial(?temp)) and
(??S.Prec = reverse(Prt_Btwn(1, |S.Rem|, S.Rem))) and
(??S.Rem = (<??temp> o ???S.Rem)))) and
(??S.Prec = (reverse(Prt_Btwn(1, |S.Rem|, S.Rem)) o
Prt_Btwn(0, 1, (<??temp> o ???S.Rem)))) and
(??S.Rem = Prt_Btwn(1, |(<??temp> o ???S.Rem)|, (<??temp>
o ???S.Rem))))
=====

```

```

(reverse(Prt_Btwn(1, |S.Rem|, S.Rem)) o Prt_Btwn(0, 1,
(<??temp> o ???S.Rem))) = reverse(S.Rem)

```

*VC 0_5 Generated from Explicitly Functional List
Specification*

A first step in attempting to prove the second VC (replacing the left side of the goal with ?S.Prec) actually results in the goal from the original VC. Of course, since the assumptions are different in the two VCs, this may not mean much. There are various ways to proceed in the proof and it is difficult to say if it would or would not be easier to prove. In fact, it could end up requiring knowledge of Prt_Btwn which might move it into the most difficult category.

IV. CONCLUSIONS

Various VCs have been tested, categorized, and re-worked. Using the minimalist prover for the implicit specification resulted in one VC (of eight VCs) that cannot be proven automatically. The explicit specification using Prt_Btwn avoided that problem VC and introduced no new difficulty for any other VC.

After attempts to prove the VCs automatically, we have seen that explicit specification can make a positive difference in the ease of proving VCs automatically, and certainly does not make the proving process more complex.

ACKNOWLEDGMENT

We thank Hampton Smith for his help with the automated prover. Thanks to Joan Krone and Bill Ogden for their insights in noting the potential verification benefits of explicit specifications.

REFERENCES

1. Kirschenbaum, J., Adcock, B., Bronish, D., Weide, B.W., Smith, H., Harton, H., Sitaraman, M., "Verified Component-Based Software: Deep Mathematics or Simple Bookkeeping?", Submitted to 11th International Conference on Software Reuse.
2. Harton, H., Krone, J, and Sitaraman M.: Formal Program Verification, Wiley Encyclopedia of Electrical and Electronics Engineering, Software Engineering Volume, John Wiley & Sons, 2008.
3. Smith, H., "Experimentation with a Minimalist Prover", to appear in Proc. RESOLVE 2009.

APPENDIX

1. List_Template specification with implicit functional specification:

```

Concept One_Way_List_Template(type Entry);
  uses Std_Integer_Fac, Modified_String_Theory;
  Type_Family List_Position is modeled by Cart_Prod
    Prec, Rem: Str(Entry);
  end;
  exemplar LP;
  initialization
    ensures LP.Prec = empty_string
    and LP.Rem = empty_string;

Oper Advance(upd LP: List_Position );
  requires LP.Rem /= empty_string;
  ensures LP.Prec o LP.Rem = #LP.Prec o #LP.Rem and
    |LP.Prec| =|#LP.Prec| + 1;

Oper Rem_Length(rest LP: List_Position): Integer;
  ensures Rem_Length = (|LP.Rem|);

Oper Insert( alt E: Entry; upd LP: List_Position );
  ensures LP.Prec = #LP.Prec and
    LP.Rem = <#E> o#LP.Rem;

Oper Remove( rpl R: Entry; upd LP: List_Position );
  requires LP.Rem /= empty_string;
  ensures LP.Prec = #LP.Prec and
    #LP.Rem = <R> o LP.Rem;

end One_Way_List_Template;

```

2. The altered List_Template using Prt_Btwn for explicit specification:

```

Concept One_Way_List_Template( type Entry);
  uses Std_Integer_Fac, Modified_String_Theory;
  Family List_Position is modeled by Cart_Prod
    Prec, Rem: Str(Entry);
  end;
  exemplar P;
  initialization
    ensures P.Prec = empty_string and
    P.Rem = empty_string;

Oper Advance( upd P: List_Position );
  requires P.Rem /= empty_string and
    Prt_btwn(0, 1, P.Rem);
  ensures P.Prec = #P.Prec o Prt_Btwn(0, 1, #P.Rem)
    and P.Rem = Prt_Btwn(1,|#P.Rem|, #P.Rem);

Oper Rem_Length(rest P: List_Position): Integer;
  ensures Rem_Length = (|P.Rem|);

Oper Insert( clr New_Entry: Entry; upd P: List_Position);
  ensures P.Prec = #P.Prec and
    P.Rem = <#New_Entry> o #P.Rem;

Oper Remove( rpl Entry_Removed: Entry;
  upd P: List_Position );
  requires P.Rem /= empty_string and
    Prt_Btwn(0, 1, P.Rem);
  ensures <Entry_Removed> = Prt_Btwn(0, 1, #P.Rem)
    and P.Prec = #P.Prec and
    P.Rem = Prt_Btwn(1,|#P.Rem|, #P.Rem);

end One_Way_List_Template;

```

Summary of VC Categorization for Both Specification Styles

	A	B	C & D
Implicit	0	1	7
Explicit	1	1	6

Note: For some provers, there is a necessary distinction between categories C and D because there are some definitions built in to the prover. However, for our prover, there is no distinction.