Towards Automated and Modular Verification in the Presence of Objects that Share Space

Yu-Shan Sun
School of Computing
Clemson University
Clemson, SC 29634
Email: yushans@clemson.edu

Abstract—Specifying and automatically verifying components with shared space in RESOLVE remains a challenge. Using a modified version of Commually_Bounded_List_Template, this paper aims to provide an insight on how the verification system must be updated in order to verify programs that use these kinds of components. New constructs and notations must be carefully specified in order for automation to be possible.

Index Terms—Components, data abstraction, formal specification, linked data structures, verification

I. INTRODUCTION

The current state of the RESOLVE verification system allows verification conditions (VCs) to be generated for components built from array-based data structures [1]. However, when components are allowed to share a common space, the verification system must be updated in order to verify programs that use this kind of components. This requires new specification constructs and proof rules that allow us to extend the capabilities of the verification system.

In order to illustrate the changes required, the Alt_CB_List_Template (ACBLT) is a List specification that sets a maximum number of elements that can be stored across all initialized List objects. Rather than using a global state variable, like Remaining_Cap in the original Communally_Bounded_List_Template (CBLT), ACBLT uses new specification constructs to achieve the same result. Abridged versions of CBLT and ACBLT are included in the appendices for reference.

II. DISCUSSION

The ACBLT contains a Max_Capacity value provided by the user and requires it to be strictly greater than 0. The abstract type List is a Cartesian product of two mathematical strings: Prec and Rem. To facilitate and simplify the specifications, the Len definition takes in a List and returns the length summation of the Prec and Rem strings.

In order to capture the total amount of space used by the List objects, the specification uses the definition variable (Def. Var) construct to define Total_Size. When objects are inserted or removed from a List, it affects the Total_Size available to all List objects.

The Total_Size definition uses the specification notations: Val_in, Receptacles and recp defined in [2][3]. Notice that this definition variable takes all the List.Receptacles into account, which are List objects that are currently initialized (and not finalized) in scope. The val_in function allows us to retrieve the mathematical type of List and pass it to the Len function. Total_Size then can be obtained by adding up all the List lengths.

When a new List object is initialized, there are no changes to Total_Size. However, when a List object is finalized, Total_Size is affected. The type specification also has a constraint that the Total_Size is less than or equal to Max_Capacity.

Although ACBLT provides multiple operations, only Insert, Remove and Occupied_Size are shown. Insert requires that Total_Size is strictly less than Max_Capacity and since there is a change to P’s size, Total_Size is affected. Remove requires that there is something to remove and after removing an element from the List, it will affect Total_Size.Occupied_Size returns the current value of Total_Size. Since there are no changes made to the shared space, Total_Size is preserved.

To verify a program like the one shown in Listing 1, the verification system will need to ensure the different values for Total_Size are provided as givens. This means creating a verification system with new proof rules that can verify components that use global variables (ex: CBLT), use definition variables (ex: ACBLT) or a combination of both to achieve the goal of sharing space. Preliminary results from applying the proof rules manually have provided simple VCs that can be proven with minimal effort.

Listing 1. Sample Program that Uses ACBLT

```
Facility List_Fac is Alt_CB_List_Template(Integer, 2) realized by ...
;
Operation Main();
Procedure
Var L1, L2, L3: List;
Var I, J: Integer;
I := 5;
J := 3;
Insert(I, L1);
Insert(J, L2);
Remove(I, L1);
Insert(I, L3);
end Main;
```

III. Conclusion

Automated verification of components with shared space remains a challenge. The new specification notations and constructs must be added to the language and appropriate proof rules must be implemented in the VC generator.

Acknowledgments

This research is funded in part by U. S. National Science Foundation grants CCF-1161916 and DUE-1022941. Any opinions, findings, conclusions, or recommendations expressed here are those of the author and do not necessarily reflect the views of the NSF. I would like to acknowledge the members of the RESOLVE/Reusable Software Research Groups at Clemson and Ohio State. Special thanks are due to Murali Sitaraman, Bill Ogden and Joan Krone for their suggestions for improvements.

References


APPENDIX A

Concept Communally_Bounded_List_Template{
  type Entry;
  evaluates Max_Capacity: Integer);
uses String_Theory, Integer_Ext_Theory;
requires Max_Capacity > 0;

Global Shared State
Abstract_Var Remaining_Cap: N;
initialization
ensures Remaining_Cap = Max_Capacity;
end;

Type Family List is modeled by Cart_Prod
Prec, Rem: Str(Entry);
end;
exemplar P;

initialization
ensures P.Prec = Empty_String and
P.Rem = Empty_String;
finalization
affects Remaining_Cap;
ensures Remaining_Cap = #Remaining_Cap + |P.Prec o P.Rem|;
constraint 0 ≤ Remaining_Cap;
end;

Operation Insert( clears New_Entry: Entry;
updates P: List );
affects Remaining_Cap;
requires Remaining_Cap > 0;
ensures P.Prec = #P.Prec and
P.Rem = <#New_Entry> o #P.Rem and
Remaining_Cap = Remaining_Cap - 1;

Operation Remove( replaces Entry_Removed: Entry;
updates P: List );
affects Remaining_Cap;
requires P.Rem ≠ Empty_String;
ensures P.Prec = #P.Prec and
Entry_Removed = DeString(
Prt_Btwn(0,1,#P.Rem)
) and
P.Rem = Prt_Btwn(1,|#P.Rem|,#P.Rem) and
Remaining_Cap = Remaining_Cap + 1;

Operation Rmng_Capacity(): Integer;
ensures Rmng_Capacity = Remaining_Cap ;

*** (Omitted) ***
end Communally_Bounded_List_Template;

Listing 2. Communally Bounded List Version 1 (CBLT)

APPENDIX B

Concept Alt_CB_List_Template{
  type Entry;
  evaluates Max_Capacity: Integer);
uses String_Theory, Integer_Ext_Theory;
requires Max_Capacity > 0;

Definition Len(P: List) : N = |P.Prec| + |P.Rem|;

*** (Theorems/Corollary/Lemmas) ***

Type Family List is modeled by Cart_Prod
Prec, Rem: Str(Entry);
end;
exemplar P;

Def. Var Total_Size : N = \sum_{S \in List.Receptacles} Len(List.Val_in(S));

initialization
ensures P.Prec = Empty_String and
P.Rem = Empty_String;
finalization
affects Total_Size;
constraint Total_Size ≤ Max_Capacity;
end;

Operation Insert( clears New_Entry: Entry;
updates P: List );
affects Total_Size;
requires Total_Size < Max_Capacity;
ensures P.Prec = #P.Prec and
P.Rem = <#New_Entry> o #P.Rem;

Operation Remove( replaces Entry_Removed: Entry;
updates P: List );
affects Total_Size;
requires P.Rem ≠ Empty_String;
ensures P.Prec = #P.Prec and
Entry_Removed = DeString(
Prt_Btwn(0,1,#P.Rem)
) and
P.Rem = Prt_Btwn(1,|#P.Rem|,#P.Rem);

Operation Occupied_Size(): Integer;
ensures Occupied_Size = Total_Size ;

*** (Omitted) ***
end Alt_CB_List_Template;

Listing 3. Communally Bounded List Version 2 (ACBLT)