Automated Verification With and Without Reference Behavior

Yu-Shan Sun and Murali Sitaraman

Technical Report RSRG-16-02
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

May 2016

Automated Verification With and Without Reference Behavior

Yu-Shan Sun and Murali Sitaraman

School of Computing, Clemson University, Clemson, SC 29634, USA
{yushans, murali}@clemson.edu

Abstract. Automated verification of software built from data abstraction is rendered difficult by reference behavior both in the client code and implementation code. In the client code, object encapsulation through component development in modern programming languages remains a problem, because clients can violate the abstraction boundary by accessing object internals through aliased object references. In data abstraction implementation code, references are often explicit and verification needs to account for them. To illustrate the solutions to the problem, this paper employs a classical list reversal example. For the client side problem, the proposed solution utilizes an abstract list interface specification that, by design, avoids the need for explicit references and aliasing. For internal code based explicitly on references, such as in place list reversal, the proposed solution involves the use of a concept that captures acyclic linked structures. Both solutions rely on standard logic and are shown to be verified automatically, using the same machinery.

Keywords: Components, data abstraction, formal specification, linked data structures, verification

1 Introduction

For software development to scale up, software needs to be built from (reusable) components. For verification to be modular and scale up, components must have specifications. These specifications will be used in reasoning about client code. A common problem in verifying such client code for data abstractions is the cross boundary aliasing problem. This paper illustrates that by careful interface design, aliasing can be avoided and automated verification can be simplified.

Reasoning about reference behavior, however, cannot be avoided in verifying lower-level implementations of data abstractions that make explicit use of references. This paper illustrates how code involving references can be verified automatically, using an interface that captures reference behavior.

Specifications of code with and without explicit reference behavior use only standard logic. The underlying verification process is the same for both cases. The ideas are illustrated using two versions of code to reverse a list, one based on a list abstraction where references are entirely hidden and another where references are explicit.

Section 2 of the paper contains the most related work. Section 3 contains a solution that avoids cross boundary aliasing. Solution in Section 4 is based on explicit references. The last section contains our conclusions.

2 Related Work

There are two scenarios of aliasing that complicate reasoning: References that are aliased **within** a component and aliased **across** components [1].

The verification system must provide a way to deal with aliasing across components when the programming languages allow references to be aliased as stated by Filipović, et al, Leavens, et al and O'Hearn, et al. [2][3][4]. There have been several different proposals to address this problem and Hatcliff, et al and Hogg, et al both presented summaries of these ideas in [5][6]. The solution proposed in this paper avoids cross-boundary aliasing by careful interface design and by the use of the swap operator instead of assignment [7].

Separation logic is an extension of Hoare logic used to reason about references within a component [8]. Like the name suggests, if the heap can be safely separated into disjoint sections and each reference only operates on a disjoint section, then it makes it possible to formally reason about the program. The following is an example code for (in-place) list reversal. (The `[e]` notation is used to denote the contents at address `e`)

```
j := nil; while i ≠ nil do
   (k := [i + 1]; [i + 1] := j; j := i; i := k).
```

Listing 1. List Reversal Example Reproduced from [8])

Separation logic introduces a *separating conjunction* logical operator of the form $P * Q$ to indicate that P and Q are from *disjoint* regions in the heap. Using this operator, the loop invariant for the code can be written as:

$$\exists \alpha, \beta. \text{list } \alpha \text{ i} * \text{list } \beta \text{ j} \wedge rev(\alpha_0) = rev(\alpha) \cdot \beta$$

where α and β are sequences referenced by `i` and `j` respectively, `list` α `i` is a linked list predicate by induction on the length of α, rev is a mathematical function for reverse and $\cdot$ denotes the concatenation operator.

The pre- and post-condition for the list reversal code can be written as `list` α_0 `i` and `list` $rev(\alpha_0)$ `j`. Even if there are other lists in scope, the *frame rule* can be used to infer that other lists are not modified [4]. Automated reasoning of programs that use separation logic can be found in: [9–12]. Inference of separation logic assertions is the topic of [11].

Dynamic frames and region logic are other efforts for generating proofs that addresses the frame problem for shared and encapsulated references [13, 14]. Both depend on the idea of an infinite set of locations (*Loc*), where each subset of locations is a *region*. An expression E is *framed* by E if it only depends on the locations in this region. If all values corresponding to the locations in the region are not modified, then E is unchanged. Both efforts include constructs to

indicate if a region is *read* or is *modified* and have ways to define the *accessibility* of references.

The alternative solution proposed in the paper for handling internal references is based on an interface that captures acyclic reference behavior, such as in linked structures. Prior work in this area can be found in [15].

3 Facilitating Client-Side Reasoning without References through Data Abstraction

Mathematical abstraction can be used to facilitate reasoning and avoid using explicit references. To illustrate the ideas, we use a generic list data abstraction and present it in RESOLVE, an imperative and object-based programming language, specification language combine with an extensible mathematical universe [16] and a modular verification system [17, 18].

Listing 2 shows a `List_Template` concept parametreized by parameteric type `Entry`. The`uses` clause imports `String_Theory` that contains mathematical string notations, such as one for string concatenation (that are used to specify the `List` abstraction) and results involving those notations, such as concatenation is associative (that are used in the verification process).

```
Concept Globally_Bounded_List_Template(type Entry);
   uses String_Theory;

   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;
   end;

   Operation Insert(alters New_Entry: Entry; updates P: List);
      ensures P.Prec = #P.Prec and
              P.Rem = <#New_Entry> o #P.Rem;

   Operation Remove(replaces Entry_Removed: Entry;
                 updates P: List);
      requires not(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 Advance(updates P: List);
      requires not(P.Rem = Empty_String);
      ensures P.Prec = #P.Prec o Prt_Btwn(0, 1, #P.Rem) and
              P.Rem = Prt_Btwn(1, |#P.Rem|, #P.Rem);
```

```
Operation Advance_to_End(updates P: List);
   ensures P.Prec = #P.Prec o #P.Rem and
           P.Rem = Empty_String;

Operation Is_Rem_Empty(restores P: List): Boolean;
   ensures Is_Rem_Empty = ( P.Rem = Empty_String );

...
```

Listing 2. Specification of a `List` Data Abstraction

A suitable mathematical conceptualization for a `List` is a pair of mathematical string of entries [19]: `Prec` and `Rem`. The current position of the list cursor can be imagined to be in between the two strings and that is where insertions and deletions can take place. The `Prec` string contains entries before the current cursor position, while the entries after the cursor are in `Rem`. An `exemplar` presents an example list `P` that is used by the `initialization ensures` clause to establish that both the `Prec` and `Rem` strings are initially empty.

Following the mathematical model, specifications of some of the`List` operations are given; Others are omitted for brevity. In the specifications, the pre-conditions (`requires` clause) and the post-conditions (`ensures` clauses) are strictly mathematical. Together, they create a contract that must be adhered to when implementing or using the operation.

The `Insert` operation specifies that the new entry is concatenated to the beginning of the incoming or old `Rem` string (denoted by #Rem). Here, `<...>` denotes a singleton-string. The parameters for `Insert` include two `specification parameter modes`, which explicitly state the effect of the operation on the parameter on exit. The `alters` mode allows `New_Entry` to pass a meaningful value to the operation, but the value of it at the end of the operation is unspecified. The motivation for this specification is to avoid cross-boundary aliasing; no copying of references or values is necessary with this specification [7]. The `updates` mode indicates that `P` had some meaningful value when passed to the operation and will be updated according to the `ensures` clause.

The `Remove` operation has the opposite effect of `Insert`. It has the opposite effect of `Insert` and it `replaces` its parametric entry with the removed entry; again, no copying takes place, avoiding aliasing. The operation requires that the `Rem` string is not empty. The `ensures` clause uses the `Prt_Btwn` to specify a substring of the original string over the specified interval, the `DeString` function that is the inverse of `<...>` and the `|...|` function that returns the length of a string. The figure below illustrates the meaning of `Prt_Btwn` with an incoming `Rem` string of <10, 20, 30>.

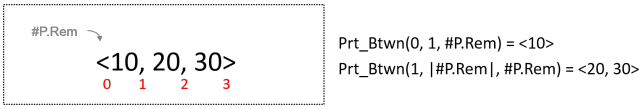

Fig. 1. Applying `Prt_Btwn`

The `Advance` operation moves the conceptual insertion point. This operation requires that `Rem` is not empty. `Advance_to_End` moves the conceptual cursor to the end of the list. `Is_Rem_Empty` returns `Boolean` to indicate whether or not the `Rem` string is empty. The parameter `P` uses the `restores` mode to indicate it had some meaningful value and will be fully restored to the value passed in.

In addition to the operations discussed explicitly, the RESOLVE language includes an implicit swap operator swap operator (`:=:`) on every type. Swapping allows data to be efficiently moved, even in the case of objects, without introducing aliasing. A comparison of (deep and shallow) copying and swapping can be found in [7].[1] The interface design, with use of parameter modes, such as `alters`, ensures that each object in the heap has exactly one reference. This approach avoids the need to explicitly separate the heap and state every modification to the heap for every operation.

3.1 List Reversal

An extension to the `Concept` can be created to add additional operations that are not provided. In this case, we define a `List_Reversal_Capability` extension that contains a `Reverse_List` operation. Here, `Reverse` is a mathematical string reversal operator.

```
Enhancement List_Reversal_Capability for
                                  Globally_Bounded_List_Template;
   Operation Reverse_List(updates L: List);
      requires L.Prec = Empty_String;
      ensures L.Prec = Reverse(#L.Rem) and L.Rem = Empty_String;
end List_Reversal_Capability;
```

Listing 3. Specification of a List Reversal Operation

The listing below is an implementation of Listing 3. This `Realization` implements `List_Reversal_Capability` using `List_Template`'s operations. This implementation uses a temporary `List` to assist the list reversal process. The loop simply removes the first entry from `L`'s `Rem` string and insert it to `Temp_List`'s `Rem` string until `L`'s `Rem` string is empty. The invariant states that `Temp_List`'s `Prec` does not change and that the incoming `L.Rem` string is the reverse of `Temp_List.Rem` string concatenated with the current `L.Rem` string. After the loop, a call to `Advance_to_End` is used to transfer the contents to `Prec`. Lastly, we swap the contents of `Temp_List` with `L` by using the swap operator (`:=:`).

```
Realization Iterative_List_Reversal_Realiz for
   List_Reversal_Capability of Globally_Bounded_List_Template;

   Procedure Reverse_List(updates L: List);
      Var Temp_List: List;
      Var Next_Entry: Entry;
```

[1] This is the same reason why C++ STL containers include a swap operator and a move operator is included in the new version.

```
      While ( not Is_Rem_Empty(L) )
         changing Temp_List, L, Next_Entry;
         maintaining Temp_List.Prec = #Temp_List.Prec and
                     Reverse(Temp_List.Rem) o L.Rem = #L.Rem;
         decreasing |L.Rem|;
      do
         Remove(Next_Entry, L);
         Insert(Next_Entry, Temp_List);
      end;
      Advance_to_End(Temp_List);
      L :=: Temp_List;
   end Reverse_List;

end Iterative_List_Reversal_Realiz;
```

Listing 4. Iterative List Reversal Implementation

Reasoning about the `Reverse_List` code listed above is straightforward using the mathematical abstraction for `List` as well as specifications for the `Advance_to_End`, `Insert`, `Is_Rem_Empty` and `Remove` operations. However, each implementation of `List` will also need to be verified, but can be done separately. A loop is used in the implementation above, however reasoning can also be done on a recursive implementation of `List` reversal.

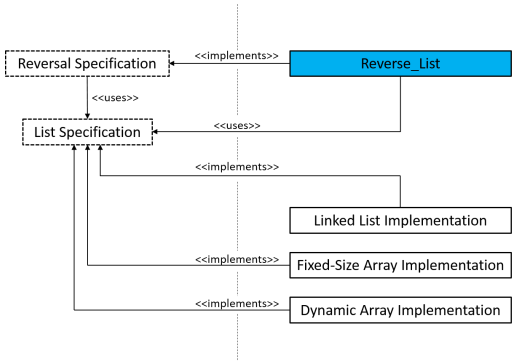

Fig. 2. Modular Reasoning with Data Abstraction

Figure 2 depicts how modular reasoning for this implementation uses the specifications facilitated by list and reverse specifications. The dotted line in the middle of the figure separates mathematical specifications from programming implementations. Even if the underlying implementation of the `List` abstraction is based on linked references, the references are hidden in the `List` implementation and do not show up in `Reverse_List`. It's specification-based reasoning that allows us to swap different implementations of `List` without having to reason about the correctness of the code for `Reverse_List` again. However, each implementation of `List` must be verified using the `List` spec-

ification and their underlying implementation's specifications. The reasoning of `Linked List Implementation` will use reference specifications, while reasoning of `Fixed Array Implementation` and `Dynamic Array Implementation` will use array specifications.

3.2 Clean Semantics

To reason through the code as we have done in Listing 4, the language in question must have *clean semantics*. There are two properties of a clean language: it's *variable-based* and *effect-restricted*. Clean semantics ensures that the specifications only deal with abstract values of variables (not their internal references) that are accessible in the current state of the operation. Any operation call's effect is restricted to the explicit parameter variables and any global variables declared to be affected. A programming language with clean semantics where one can *directly* reason without involving references is the topic of [20].

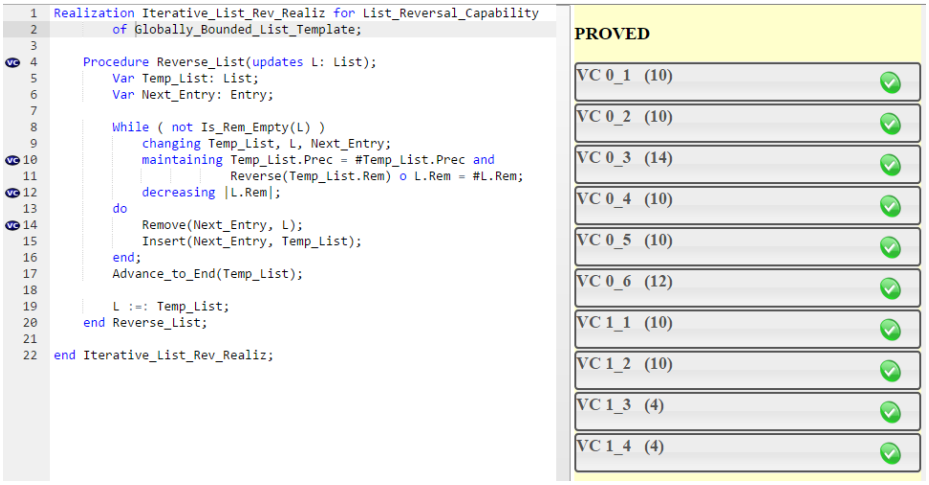

Fig. 3. Verification of Listing 4 Using an Automated Prover

3.3 Automated Verification

The RESOLVE verifying compiler is motivated by the verification grand challenge [21], and it uses a set of proof rules to mechanically generate VCs that are necessary and sufficient to prove the correctness of Listing 4 [17]. For the present example, the system generates 4 VCs to establish the base case of the loop invariant, 2 VCs to establish the inductive case of the loop invariant, 1 VC for the requires clause of `Remove`, 1 VC to show termination of the while loop and 2 VCs for the post-condition of `Reverse_List`. The underlying prover has a sophisticated uninterpreted function solver, tailored to handle sequents, and is under continued development; an earlier version is discussed in [18]. The figure below shows this new prover in action using web integrated environment that

has been used in classrooms [22]. All the VCs are proven in a few seconds by this new automated prover. The proof process is sound (meaning no false assertions would be proved), but necessarily incomplete (meaning some true assertions will not be proved).

4 Reasoning with References Explicitly

This section contains a version of list reversal (similar in spirit to the one in Section 2) and is based on explicit use of references. It is based on a concept that captures acyclic reference behavior that includes a type `Pos`, a pointer to some `Info` type. The formal concept along with specifications of global state, denoted by `Content` and `Ref`, operations to manipulate references and contents, such as `Redirect_Ref_at` and `Is_Void` are detailed in the next section.

```
Operation Reverse_Linked_List(clears i: Pos; updates j: Pos);
   affects Content, Ref;
   requires j = Void;
   ensures Info_String(Content, Ref, #i) =
           Reverse(Content, Ref, Info_String(j));
Procedure
   While ( not Is_Void(i) )
      maintaining Info_String(Content, Ref, #i) =
                  Reverse(Info_String(Content, Ref, j)) o
                  Info_String(Content, Ref, i) and
                  Are_Disjoint_Refs(Ref, i, j);
      decreasing Distance_to_Void(Ref, i);
   do
      Redirect_Ref_at(i, j);
      i :=: j;
   end;
end Reverse_Linked_List;
```

Listing 5. List Reversal Using Explicit References

The `Redirect_Ref_at(i, j)` and `i :=: j` statements are equivalent to the listing below. `k` and `m` are newly created reference variables and `i->next` is the reference that is pointed by `i`'s next field. By swapping `i` and `j`, it performs in-place reversal of a list like the one in Listing 1.[2]

```
k = i->next; i->next = j; j = k; // Redirect_Ref_at(i, j);
m = i; i = j; j = m;             // i :=: j;
```

Listing 6. Informal Explanation of the Above Code

The figure below illustrates how the reversal occurs for each iteration of the loop with different references of `i` and `j`, which is of type `Pos`. `nil` is a special reference location and the `clears` mode indicates the outgoing value of `i` set to `nil`.

[2] The code could have also been written more verbosely using other operations such as `Follow_Ref` and `Are_Colocated` from UVRT.

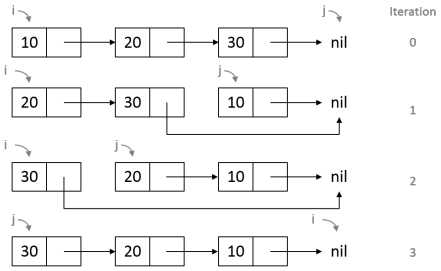

Fig. 4. Illustration of the Working of the Code

The definition `Info_String` takes a location and constructs a mathematical string from reachable memory addresses' content using the mathematical functions `Content` and `Ref` (These are defined in Section 4.1).

The loop invariant states that the string of contents in the incoming location `i` is the same as the reverse of the contents in `j` concatenated with the string of contents in the location `i`. It also states that all the closure of locations obtained from applying the `Ref` function to `i` is disjoint from the closure of all locations from applying `Ref` to `j`. This invariant is equivalent to the loop invariant from Section 2, however it avoids the need of using the $*$ operator. Both disjointness and reversal can be stated with standard logic. Using Figure 4, we can see that the loop invariant holds for all iterations.

The `decreasing` clause states that this loop terminates. The definition `Distance_to_Void` returns the distance to `nil`. This definition can be used by the clause to claim that location `i` is moving closer to `nil` after each iteration of the loop. Once `i` is equal to `nil`, the loop condition no longer holds and the loop is terminated.

4.1 Ultimately Void Referencing Template

`Ultimately_Void_Referencing_Template` (`UVRT`) is a specialized version of a component that captures reference behavior that is especially suitable for implementing non-cyclic structures. In order to better explain `UVRT` specification, the concept has been broken down to smaller sections.

```
Concept Ultimately_Void_Referencing_Template(type Info);
  uses Function_Theory with Terminal_Range_Op_Ext;

  Defines Location: 𝕊et;
  Defines Void: Location;

  Var Ref: Location –> Location;
  Var Content: Location –> Info;
   constraints Terminal_Range(Location,{Ref},Location) ⊆ {Void}
                                  which_entails Ref(Void) = Void;
   ...
```

Listing 7. `UVRT` (Shared State)

For the formal function definitions and notations used in the specification, the concept in Listing 7, makes use of `Function_Theory` and its extensions. In the listing, the `Location` set is an abstraction of the address space and its actual size is defined and constrained by an implementation on the underlying machine. `Void` is a special `Location`. A key idea in the concept is the use of two global state variables to capture the shared state: `Ref`, a function that gives the "next" location for a given location and `Content`, a function that gives the information value referenced by a given location. Figure 5 depicts how the global state variables captures the share state.

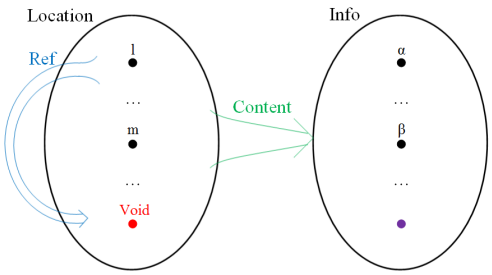

Fig. 5. Global State Variables `Ref` and `Content`

`UVRT` specification has been designed with the goal of easing automated verification. For example, through carefully defined mathematical notations and development of results, it avoids the use of quantifiers.

4.1.1 Absence of Cycles In Listing 7, the key constraint is that following the next `Ref` chain for every location, will ultimately reference (or reach) the `Void` location. This constraint is the basis for the name for the concept and ensures that there are no cycles. Since this constraint is already a given, when we implement Stacks, Queues, Lists (or Trees with a n-reference generalization) using `UVRT`, it becomes a freely established representation invariant that requires no further proof.

In order to express the constraint formally, we use a mathematical definition `Terminal_Range`. For this constraint, there is only one function `Ref` that is applied to the set `Location` to determine the terminal range which is restricted to be just `Void`. The *which_entails* clause gives a lemma (that needs to be proved and) that becomes a useful lemma in the automated verification process. This ensures that the result of applying `Ref` to `Void` is simply `Void`. Figure 6 provides an illustration of this definition.

In verifying shared `List` or other realizations that are based on `UVRT`, verification conditions involve `Terminal_Range`, and these will be discharged by an automated prover using pre-established theorems in `Terminal_Range_Op_Ext`.

4.1.2 Absence of Memory Leaks In Figure 8, `UVRT` defines a programming type `Pos` to represent a reference, mathematically modeled as a `Location`. Initially, each position takes the value `Void`. (`exemplar` is just an example value of

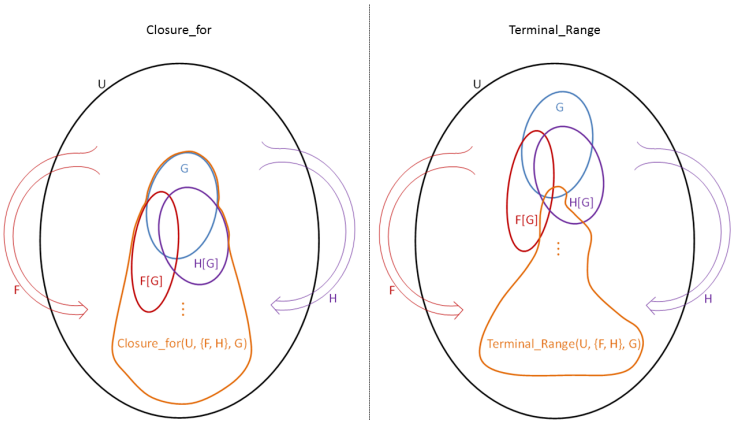

Fig. 6. Pictorial Explanations of `Closure_for` and `Terminal_Range`

type `Pos`) A second key idea is "accessibility" and it is specified by a variable mathematical definition `Accessible_Loc`, whose value depends upon the global state variable `Ref`.

```
Type Family Pos is modeled by Location;
  exemplar p;
  initialization ensures p = Void;

  Def Var Accessible_Loc: 𝒫(Location) = (
    {Void} ∪ Closure_for(Location, {Ref},
                                    Pos.Val_in[Pos.Receptacle]));

  finalization
    affects Ref, Content, Accessible_Loc;
    ensures ...
  constraint ...
end;
```

Listing 8. `UVRT` (Type Definition)

The formal definition of `Accessible_Loc` is based on a mathematical definition `Closure_for`, also defined and elaborated in `Terminal_Range_Op_Ext` theory. The `Closure_for(U, {F, H}, G)` returns a set that results from applying the functions `F` and `H` repeated to the set of elements in `G`; `G` is a subset of `U`. The construct $[\ldots]$ allows for a function f to be applied repeatedly to the *Powerset* of the domain D and returns the *Powerset* of the range R. Here, `Accessible_Loc` is the set of reachable locations produced by the `Closure_for` on all programming variables of a `Pos` type (i.e., all void-referencing reference variables), unionized with `Void`.

In the definition of `Accessible_Loc` as well in the specification of operations, the following notations are used. They are a part of the specification language

that allow us to make assertions about all objects or a specific object of a certain type or refer to the actual programming variable associated with a name.

- T.**Receptacle** denotes the set of all variables of type `T` that have been initialized, but not finalized
- **recp**.p is a specification language construct, it refers to the actual variable that will be associated with `p`
- **Val_in** (**recp**.p) denotes the mathematical value corresponding to the receptacle `p`

The finalization of a UVRT position variable (or reference) – not shown for brevity – will have to deal with two scenarios. For all locations in `q` $\in$ `Accessible_Loc` that are accessible from the set of all allocated locations minus `p` (the pointer that is being finalized or removed), then no changes are done to their references, i.e., `Ref(q) = Ref(#q)`. However, if some `q` is no longer accessible, because of the finalization of `p`, then `q` becomes available for allocation. In other words, every location is either available for allocation or is accessible, i.e., there are no memory leaks. In other words, the specification of UVRT demands that the underlying implementation of it do garbage collection. Using the new `Accessible_Loc`, the specification also states that that the `Content` prior to finalizing `p` is equal to the `Content` after finalization for accessible locations.

4.1.3 UVRT Operations The rest of the UVRT operations are discussed here briefly. `Give_New_Loc` allocates an unused location for a new `Pos`; it is the equivalent of memory allocation. `Redirect_Ref_at` makes referent point towards what `Ref(p)` points to. Operation `Follow_Ref` moves `p` to the reference pointed by `p`. Finalization for the original `p` will be in effect after this operation is called. `Swap_Content_of` swaps the information pointed at `p` with `I`. `Relocate_to` replace `p` with `New_L` and contents of old `p` is finalized; also unaccessible locations are specified to be free as in finalization. `Are_Colocated` checks if two `Pos` point to the same memory location. `Is_Almost_Inaccessible` checks if `p` can be accessible from other `Pos` other than `p`. `Is_Void` checks if a `Pos` is `Void`. `Set_to_Void` sets a `Pos` to `Void` and finalizes all resources. Listing 9 shows specifications for `Give_New_Loc`, `Redirect_Ref_at` and `Is_Void`, due to space constraints the rest of operations are omitted. The `Eq_Except_On` function allow us to state that all references remains the same, except at `Ref(p)`.

```
Operation Give_New_Loc(updates p: Pos);
   affects Accessible_Loc;
   requires p = Void;
   ensures p ∉ #Accessible_Loc;

Operation Redirect_Ref_at(preserves p: Pos,
                          updates referent: Pos);
   affects Ref;
   requires p ∉ Closure_for(Location, {Ref}, {referent});
   ensures Eq_Except_On(#Ref, Ref, p) and
           Ref(p) = #referent and referent = #Ref(p);
```

```
Operation Is_Void(preserves p: Pos): Boolean;
   ensures Is_Void = ( p = Void );
```

Listing 9. Specifications of Some UVRT Operations

4.2 Automated Verification of `Reverse_Linked_List`

`Reverse_Linked_List` uses string concatenation definition and theorems from `String_Theory` and `Info_String` and `Distance_to_Void` definition and theorems from `SS_Theory`. Figure 7 shows the automated verification results.

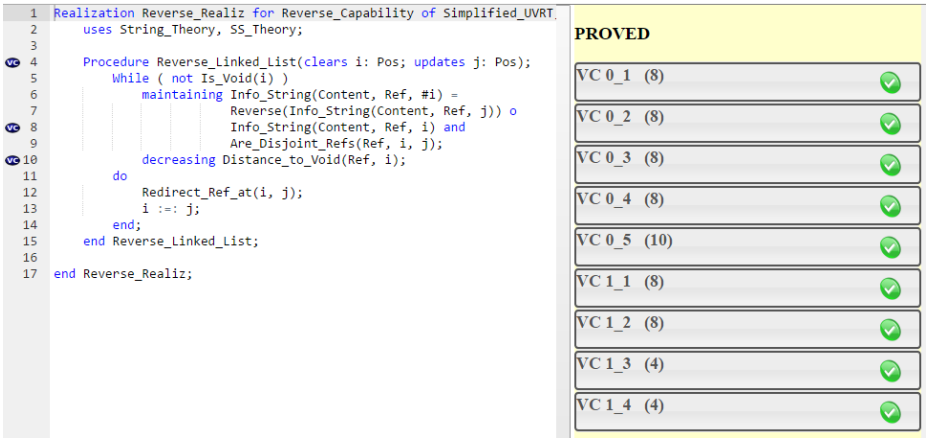

Fig. 7. Verification of Listing 5 Using an Automated Prover

The compiler generates 9 VCs that are necessary and sufficient to prove the correctness of Listing 5.[3] Our loop rule generates two paths: one where the loop condition holds and one where the negation of condition is true. This means that the base and inductive case of the invariant must hold for both possible paths, which results in 6 VCs for the loop invariant. There are two post-conditions for `Reverse_Linked_List`: the explicit `ensures` clause and one to ensure that `i` is set to the initial value of `Pos`, which is `Void`. Lastly, there must be a VC that established the loop actually terminates. As an example, the VC establishing the termination of `Reverse_Linked_List` is shown below:

Goal:
`(1 + Distance_to_Void(Ref,Ref(i'))) ≤ Distance_to_Void(Ref,i')`

Given:
1. `Eq_Except_On(Ref, Ref, i')`
2. `(Ref(i') = j')`
3. **not**`((i' = Void))`
4. `(Info_String(Content, Ref, i) =`

[3] There should be one more VC pertaining to the `requires` clause of `Redirect_Ref_at` and it should verify once the development of relevant theorems are complete.

```
   (Reverse(Info_String(Content, Ref, j')) o
   Info_String(Content, Ref, i')))
5. (j = Void)
```

Listing 10. Termination of While Loop VC

Using the definition of `Ref`, it is easy to see that the distance to `Void` of `Ref(i')` is exactly 1 less than the distance to `Void` of `i'`. Therefore, `SS_Theory` contains a theorem that will allow us to verify this VC automatically.

5 Conclusions

This paper has illustrated automated verification for software built from data abstraction components. For verification of client code, it shows, through careful interface design how aliasing can be avoided and automated verification can be simplified. For verification of code based on explicit reference behavior, it shows what is entailed in automated verification if only standard logic is employed. While we have used a simple list reversal example in this paper to illustrate the ideas, the goal of ongoing research is to develop and experiment with verification of a library of components with and without reference behavior.

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 authors and do not necessarily reflect the views of the NSF. We would like to acknowledge the members of the RESOLVE/Reusable Software Research Groups at Clemson and Ohio State. Our special thanks are due to Bill Ogden and Joan Krone for their insights in developing the specifications given in this paper.

References

1. Weide, B. W., and Heym, W.D. "Specification and verification with references". In Proc. of OOPSLA Workshop on Specification and Verification of Component-Based Systems, 2001, 50-59.
2. Filipović, I., O'Hearn, P., Torp-Smith, N., Yang, H. "Blaming the client: on data refinement in the presence of pointers". Formal Aspects of Computing 22, 2010, 547-583.
3. Leavens, G. T., Leino, K. R. M., and Müller, P., "Specification and verification challenges for sequential object-oriented programs". Formal Aspects of Computing 19, 2007, 159-189.
4. O'Hearn, P., Reynolds, J., and Yang, H. "Local reasoning about programs that alter data structures". In Proc. 15th International Workshop on Computer Science Logic (CSL 2001), LNCS 2142, Springer, 2001, 1-19.
5. Hatcliff, J., et al., "Behavioral Interface Specification Languages". Dept. of EECS, University of Central Florida, CS-TR-09-01, March 2009.
6. Hogg, J., Lea, D., Willis, A., deChampeaux, D., and Holt, R. "The Geneva convention on the treatment of object aliasing". SIGPLAN OOPS Mess. 3, 1992, 11-16.

7. Harms, D. E., and Weide, B. W. "Copying and Swapping: Influences on the Design of Reusable Software Components", IEEE Trans. Softw. Eng. 17 (5), IEEE Press, 1991, 424-435.
8. Reynolds, J. C., "Separation Logic: A Logic for Shared Mutable Data Structures", In Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002), IEEE Computer Society, 2000, 55-74.
9. Botinčan, M., Parkinson, M., and Schulte, W., "Separation Logic Verification of C Programs with an SMT Solver", Electronic Notes in Theoretical Computer Science (ENTCS), 254, 2009, 5-23.
10. Bouillaguet, C., et al., "Using First-Order Theorem Provers in the Jahob Data Structure Verification System", Lecture Notes in Computer Science, 4349, 2007, 74-88.
11. Mühlberg J. T., et al., "Learning Assertions to Verify Linked-List Programs", In Proc. of the 13th International Conference on Software Engineering and Formal Methods (SEFM 2015), 2015, 37-52.
12. Piskac, R., Wies, T., and Zufferey, D., "Automating Separation Logic Using SMT", In Proc. of the 25th International Conference on Computer Aided Verification (CAV'13), 2013, 773-789.
13. Kassios, I. "Dynamic frames: support for framing, dependencies and sharing without restrictions". In Proc. FM 2006: Formal Methods, LNCS 4085, Springer, 2006, 268-283.
14. Banerjee, A. and Naumann, D. A., and Rosenberg, S., "Regional Logic for Local Reasoning About Global Invariants", In Proc. of the 22nd European Conference on Object-Oriented Programming (ECOOP '08), Springer-Verlag, 2008, 387-411.
15. Kulczycki, G., et al., "The Location Linking Concept: A Basis for Verification of Code Using Pointers". In Proc. of the 4th international conference on Verified Software: theories, tools, experiments (VSTTE'12), 2012, 34-49.
16. Sitaraman M., et al., "Building a Push-Button RESOLVE Verifier: Progress and Challenges", Formal Aspects of Computing 23 (3), Springer, 2011, 607-626.
17. Harton, H. "Mechanical and Modular Verification Condition Generation for Object-Based Software", Ph.D Dissertation, Clemson University, 2011.
18. Cook, C. T., et al., "Specification engineering and modular verification using a web-integrated verifying compiler", In Proc. of the 34th International Conference on Software Engineering (ICSE 2012), 2012, 1379-1382.
19. Sitaraman, M., et al. "Reasoning About Software-Component Behavior", In Proc. of the 6th International Conference on Software Reuse: Advances in Software Reusability (ICSR 2000), Springer-Verlag, 2000, 266-283.
20. Kulczycki, G. "Direct Reasoning", Ph.D Dissertation, Clemson University, 2004.
21. Hoare, C. A. R., "The Verifying Compiler: A Grand Challenge for Computing Research", Journal of the ACM 50, ACM, 2003, 63-69.
22. Kabbani, N. M., et al., "Formal Reasoning Using an Iterative Approach with an Integrated WebIDE", In Proc. of the 2nd International Workshop on Formal Integrated Development Environment (F-IDE 2015), 2015, 56-71.