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

Copyright © 2016 by the authors. All rights reserved.
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)

\[
j := \text{nil}; \text{while } i \neq \text{nil} \text{ do } \newline\quad \{ k := [i + 1]; [i + 1] := j; j := i; i := k \}.\]

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

Separation logic introduces a separating conjunction logical operator of the form $P \ast Q$ 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:

\[
\exists \alpha, \beta. \text{list } \alpha \ i \ast \text{list } \beta \ j \land \text{rev}(\alpha_0) = \text{rev}(\alpha) \cdot \beta
\]

where $\alpha$ and $\beta$ are sequences referenced by $i$ and $j$ respectively, list $\alpha \ i$ is a linked list predicate by induction on the length of $\alpha$, 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 $\text{list } \alpha_0 \ i$ and $\text{list } \text{rev}(\alpha_0) \ j$. 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 ($\text{Loc}$), 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 parametrized 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).

```plaintext
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);
```
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>.

![Figure 1. Applying Prt_Btwn](image)
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 (**:=:**) 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]. 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;


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. Its 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](image)

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 IsVoid are detailed in the next section.

Operation ReverseLinkedList(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 IsVoid(i) )
   maintaining Info_String(Content, Ref, #i) =
         Reverse(Info_String(Content, Ref, j)) o
         Info_String(Content, Ref, i) and
         Are_Disjoint.Ref(Ref, i, j);
   decreasing Distance_to Void(Ref, i);
   do
      Redirect_Ref_at(i, j);
      i :=: j;
   end;
end ReverseLinkedList;

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.
The definition \texttt{Info\_String} takes a location and constructs a mathematical string from reachable memory addresses' content using the mathematical functions \texttt{Content} and \texttt{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 \texttt{Ref} function to \( i \) is disjoint from the closure of all locations from applying \texttt{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 \texttt{decreasing} clause states that this loop terminates. The definition \texttt{Distance\_to\_Void} returns the distance to \texttt{nil}. This definition can be used by the clause to claim that location \( i \) is moving closer to \texttt{nil} after each iteration of the loop. Once \( i \) is equal to \texttt{nil}, the loop condition no longer holds and the loop is terminated.

4.1 Ultimately Void Referencing Template

\texttt{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 \texttt{UVRT} specification, the concept has been broken down to smaller sections.

\texttt{Concept Ultimately\_Void\_Referencing\_Template(type Info);}  
\hspace{1cm} \texttt{uses Function\_Theory with Terminal\_Range\_Op\_Ext;}  
\texttt{Defines Location: Set;}  
\texttt{Defines Void: Location;}  
\texttt{Var Ref: Location \rightarrow Location;}  
\texttt{Var Content: Location \rightarrow Info;}  
\texttt{constraints Terminal\_Range(Location,\{Ref\},Location) \subseteq \{Void\} \hspace{1cm} \text{which entails} \hspace{1cm} Ref(Void) = Void;}  
\hspace{1cm} \ldots

\texttt{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
A second key idea is “accessibility” and it is specified by a variable mathematical definition \texttt{Accessible\_Loc}, whose value depends upon the global state variable \texttt{Ref}.

\begin{verbatim}
Type Family Pos is modeled by Location;
  exemplar p;
  initialization ensures p = Void;

Def Var Accessible\_Loc: \mathcal{P}(\text{Location}) = (\{VOID\} \cup \text{Closure\_for}(\text{Location}, \{\text{Ref}\},
  \text{Pos}\_\text{Val\_in}([\text{Pos}\_\text{Receptacle}])));

finalization
  affects Ref, Content, Accessible\_Loc;
  ensures ...
  constraint ...
end;

Listing 8. UVRT (Type Definition)
\end{verbatim}

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

In the definition of \texttt{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
- \texttt{recp.p} is a specification language construct, it refers to the actual variable that will be associated with \( p \)
- **\texttt{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 \( q \in \text{Accessible\ Loc} \) 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., \( \text{Ref}(q) = \text{Ref} (#q) \). 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 \texttt{Accessible\ Loc}, the specification also states that that the \texttt{Content} prior to finalizing \( p \) is equal to the \texttt{Content} after finalization for accessible locations.

### 4.1.3 UVRT Operations

The rest of the UVRT operations are discussed here briefly. \texttt{Give\ New\ Loc} allocates an unused location for a new Pos; it is the equivalent of memory allocation. \texttt{Redirect\ Ref\ at} makes referent point towards what \texttt{Ref(p)} points to. Operation \texttt{Follow\ Ref} moves \( p \) to the reference pointed by \( p \). Finalization for the original \( p \) will be in effect after this operation is called. \texttt{Swap\ Content\ of} swaps the information pointed at \( p \) with \texttt{I}. \texttt{Relocate\ to} replace \( p \) with \texttt{New\ L} and contents of old \( p \) is finalized; also unaccessible locations are specified to be free as in finalization. \texttt{Are\ Colocated} checks if two Pos point to the same memory location. \texttt{Is\ Almost\ Inaccessible} checks if \( p \) can be accessible from other Pos other than \( p \). \texttt{Is\ Void} checks if a Pos is Void. \texttt{Set\ to\ Void} sets a Pos to Void and finalizes all resources. Listing 9 shows specifications for \texttt{Give\ New\ Loc}, \texttt{Redirect\ Ref\ at} and \texttt{Is\ Void}, due to space constraints the rest of operations are omitted. The \texttt{Eq\ Except\ On} function allow us to state that all references remains the same, except at \texttt{Ref(p)}.

\begin{verbatim}
Operation Give_New_Loc(updates p: Pos);
    affects Accessible_Loc;
    requires p = Void;
    ensures p \notin \#Accessible_Loc;

Operation Redirect_Ref_at(preserves p: Pos,
    updates referent: Pos);
    affects Ref;
    requires p \notin Closure_for(Location, \{Ref\}, \{referent\});
    ensures EqExcept_On(#Ref, Ref, p) and
        Ref(p) = \#referent and referent = \#Ref(p);
\end{verbatim}
Operation \texttt{Is\_Void}(\texttt{preserves} p: \texttt{Pos}): \texttt{Boolean};
\hspace{1em} \texttt{ensures} \texttt{Is\_Void} = (p = \texttt{Void});

Listing 9. Specifications of Some UVRT Operations

4.2 Automated Verification of \texttt{Reverse\_Linked\_List}

\texttt{Reverse\_Linked\_List} uses string concatenation definition and theorems from \texttt{String\_Theory} and \texttt{Info\_String} and \texttt{Distance\_to\_Void} definition and theorems from \texttt{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 \texttt{Reverse\_Linked\_List}: the explicit \texttt{ensures} clause and one to ensure that \texttt{i} is set to the initial value of \texttt{Pos}, which is \texttt{Void}. Lastly, there must be a VC that established the loop actually terminates. As an example, the VC establishing the termination of \texttt{Reverse\_Linked\_List} is shown below:

\textbf{Goal:}
\begin{equation}
(1 + \texttt{Distance\_to\_Void}(\texttt{Ref}, \texttt{Ref}(i'))) \leq \texttt{Distance\_to\_Void}(\texttt{Ref}, i')
\end{equation}

\textbf{Given:}
1. \texttt{Eq\_Except\_On}(\texttt{Ref}, \texttt{Ref}, i')
2. \texttt{(Ref(i') = j')}
3. \texttt{not((i' = \texttt{Void})}
4. \texttt{(Info\_String(Content, Ref, i) =}

³ There should be one more VC pertaining to the \texttt{requires} clause of \texttt{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 $\text{Ref}$, it is easy to see that the distance to $\text{Void}$ of $\text{Ref}(i')$ is exactly 1 less than the distance to $\text{Void}$ of $i'$. Therefore, $\text{SS Theory}$ 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


