Specification and Reasoning about Shared Realizations: An Illustrative Example

Yu-Shan Sun, Diego Zaccai, and Murali Sitaraman

Technical Report RSRG-13-04
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

February 2013

Specification and Reasoning about Shared Realizations: An Illustrative Example

Yu-Shan Sun[1], Diego Zaccai[2], and Murali Sitaraman[1]

[1] School of Computing, Clemson University, Clemson, SC 29634, USA
{yushans | msitara}@clemson.edu
[2] Department of Computer and Information Science,
The Ohio State University, Columbus, OH, 43210, USA
zaccai.1@osu.edu

Abstract. While the general idea of data abstraction is well understood and most formal specification methods support their development and use in some form or the other, object encapsulation through component development in modern programming languages remains a problem, because clients can violate the abstraction by accessing object internals through aliased object references. The purpose of this paper is to illustrate how to make use of a shared representation and reference copying for efficiency in a data abstraction implementation, yet avoid clients from having to reason about aliased references. The paper illustrates the ideas with a detailed example in which a shared realization is employed to produce an (amortized cost) constant time implementation of an operation to copy a buffer. It discusses the specification and reasoning machinery necessary to prove that a realization with a shared data representation is correct with respect to an abstract specification, and shows how clients of such realizations can be verified using abstract interfaces alone.

Keywords: Formal specification, linked data structures, verification

1 Introduction

Reasoning about realizations in which a data representation is shared among objects is a challenging problem. Such shared realizations are often motivated by performance improvements. The objective of this paper is to illustrate with a simple, yet motivating example the key ideas necessary for formal verification of shared realizations. The paper additionally illustrates the use of intermediate data abstractions to simplify such verification, and help hide a host of complexities in verifying such realizations.

It is easy to create a normal mutable buffer (or queue) that has constant time operations such as Enqueue and Dequeue but whose copy would require linear time. Alternatively, immutable buffers could be implemented to have constant-time copying, through reference copying, but the performance of other operations would suffer due to the immutability of the underlying structures. The solution presented in this paper is a buffer data abstraction in which all operations, including copying, take amortized constant time.

Rather than implement the immutable queue structure directly, we have chosen to represent a queue with a pair of stacks and implement the stacks with a shared realization. This separation helps us illustrate how verification of an immutable queue implementation is vastly simplified; The paper includes results from its automated verification specified using the RESOLVE language and verified using the Ohio State RESOLVE compiler [1]. Later, we explain the specifications and assertions necessary to verify the shared stack realization; while generation of verification conditions for this more complex implementation is automated, tool work for automated discharge of those conditions remains to be done at the time of this paper submission.

Figure 1 illustrates the scope of the problem and the focus of different sections of this paper. It is a UML diagram showing relationships among the various artifacts. Starting from the top of the diagram, Immutable_Queue_Template is the specification of a queue concept that captures the spirit of an immutable queue. The figure also shows an implementation, named Two_Stack_Realization for the queue concept. This implementation is based on Stack_Template. Implementing immutable queues requires the ability to replicate a queue and the performance profile documents that the Two_Stack_Realization provides constant time performance for all queue operations. Due to the lack of space, the performance profiles (containing duration and memory usage estimates) themselves are not presented in this paper. Formal notations for writing performance profiles may be found elsewhere [10]. Section 2 of the paper presents an immutable queue concept, its realization, results from automated verification of the realization, and a discussion of its constant-time performance behavior.

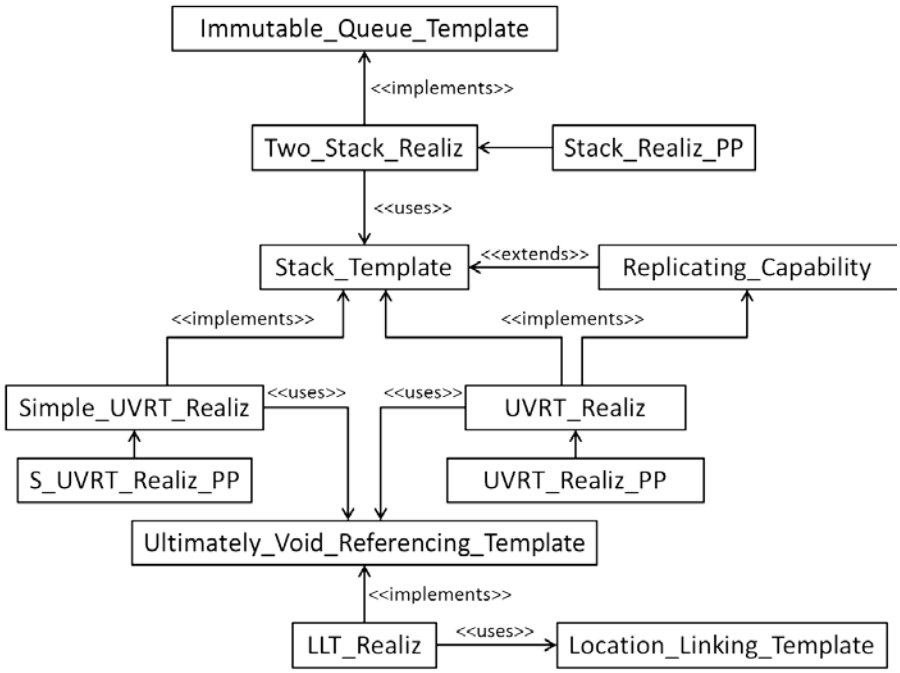

Fig. 1. An Illustrative Overview

The diagram also shows two shared realizations of Stack_Template. Both realizations use Ultimately_Void_Referencing_Template (UVRT, for short) to implement the Stack operations. The distinction between Simple_UVRT_Realiz and UVRT_Realiz is that the latter provides the ability to replicate a Stack in constant time, a feature necessary to implement immutable queue operations efficiently.

Section 3 discusses the shared stack realizations, their correctness, and the UVRT concept. UVRT is a specialized version of the general pointering concept Linked_Location_Template specified in [12] in that UVRT is constrained to allow only cycle-free pointer chains, the kind necessary for implementing structures like stacks, queues, lists, and trees. The earlier paper also illustrates how describing pointer behavior through a formally-specified data abstraction concept makes it possible to use the same verification machinery for all realizations, whether they are built using typically built-in structures, such as pointers, or they are built using other data abstractions, such as stacks. Ultimately, Linked_Location_Template will be used to realize UVRT. This realization, named LLT_Realiz in the illustration is beyond the scope of this paper and is not discussed here.

Section 4 of the paper contains related work and our conclusions are presented in Section 5.

2 Specification and an implementation of an Immutable Queue Concept

2.1 Immutable Queue specification

Figure 2 contains the specification of an immutable queue concept. An astute reader might notice that these queues are not exactly immutable: the value of a one of the queues is "replaced" in every call to Enqueue or Dequeue. However, the design is similar in the spirit of a typical object-oriented immutable queue. Specifically, the operation to Enqueue an element into the queue results in two different queues, the original one and a new one that contains all of the elements from the original one plus the one enqueued. Similarly, Dequeue does not alter the state of the queue from which the element is being removed, but instead preserves the original value of the queue and produces the element that is being removed and a new queue with the remaining elements in it. It is worth noting that the queue on which the actions are being performed are all restored, meaning that their values are unchanged; the output queues replace the actual argument queues with new values.

Some features of the language are worth mentioning here. First, the model types and values presented in specifications refer to a mathematical string of Entry, where Entry refers to the mathematical type of the parameterized object. The # in front of a parameter represents the mathematical value of the parameter at the beginning of the call. The < _ > operator is a string constructor that creates a string containing only the element inside of it. The * operator denotes string concatenation.

```
Concept Immutable_Queue_Template (type Entry);
    Type Family Queue is modeled by string of Entry
        exemplar q;
        initialization ensures q = empty_string;

    Operation Enqueue (restores q1: Queue,
                       replaces q2: Queue, clears x: Entry);
        ensures q2 = q1 * <#x>;

    Operation Dequeue (restores q1: Queue,
                       replaces q2: Queue, replaces x: Entry);
        requires q1 /= empty_string;
        ensures q1 = <x> * q2;

    Operation IsEmpty (restores q: Queue): Boolean;
        ensures IsEmpty = (q = empty_string);

end Immutable_Queue_Template;
```

Fig. 2. Specification of an Immutable Queue Contract

2.2 Two-Stack realization of immutable queues

Implementation of the queue is done with a pair of stacks. The details of the representation closely follow those presented by [15]. The queue is represented by two stacks: front and back. Unlike in [15], they are not immutable functional Lists, though to simulate immutability their values are not changed during the lifespan of a queue. The complete relationship between the abstract values of the stack and the abstract value of the queue they represent is provided by the correspondence (or abstraction) function: `q.front * reverse (q.back)`. A sample representation of a queue a
nd its corresponding value is presented in Figure 3.

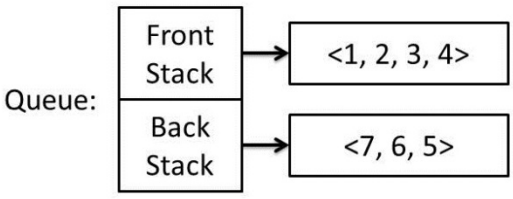

Fig. 3. Graph of the abstract value of the front and back stacks in the representation of a Queue whose abstract value is <1, 2, 3, 4, 5, 6, 7>

The back stack contains the tail of the queue in reverse order (the last element enqueued will be in the top of the stack), while the front stack contains the front of the queue in order (the top of the stack should be the first element to be dequeued). Code

for these procedures is given below. An abridged specification of Stack_Template on which this code is based is given following the implementation.

```
procedure Enqueue (restores q1: Queue,
                   replaces q2: Queue, clears x: Entry);
    q2.front := Replica (q1.front);
    q2.back := Replica (q1.back);
    Push (q2.back, x);
end Enqueue;

procedure Dequeue (restores q1: Queue,
                   replaces q2: Queue, replaces x: Entry);
    if not Is_Empty(q1.front) then
        q2.front := Replica (q1.front);
        q2.back := Replica (q1.back);
    else
        q1.front :=: q1.back
        Reverse (q1.front)
        q2.front:= Replica (q1.front)
        Clear ( q2.back )
    end if;
    Pop (q2.front, x);
end Dequeue;
```

Fig. 4. Excerpts from a two-Stack Implementation of the immutable queue

In a specification of stacks, Stacks are also conceptualized mathematically as strings of entries. Specifications of the operations mutating Stack operations Push, Pop, and Replica are straightforward and they are shown below.

```
Push(updates S: Stack; clears E: Entry);
    ensures S = ⟨#E⟩ * #S;

Pop(updates S: Stack; replaces R: Entry);
    requires S /= empty_string;
    ensures #S = <R> * S;

Replica(restores S: Stack): Stack;
    ensures Replica = S;
```

Fig. 5. Specification of Stack Operations

2.3 Automated Verification of Two-Stack Realization

In modular or specification-based verification, the implementation of a component is verified with respect to the contracts used in its representation, abstracting away all of

their implementation details. It is because of the modularity of our proof system that we can prove the correctness of our queue implementation only from the contracts used in its implementation, namely Stack_Template and Replica for Stack. Since the proof of our implementation is completely independent from that of the Stack, the proof does not hinge on the specifics of the Ultimately Void Referencing Template used to create the stack. Further, none of the VCs refer to anything other than the mathematical string modeling employed in the stack and queue specifications.

The verification of the implementation is done entirely automatically with the OSU tool-chain. The VC generator generated 12 verification conditions (VCs) for the code. Six of them were for Dequeue, 2 for Enqueue, 2 for the initialization, and 2 for Is_Empty. Most of the VCs fall into the "book keeping" category [9]. Our tool-chain allows for the use of multiple verifiers and can be connected to both Isabelle and Z3. However, due to the simplicity of the proofs, we decided to only use our in-house automatic verifier Split Decision [2]. All of the VCs were verified in less than 1 minute.

1: $q1.front_0 \circ reverse\ (q1.back_0) \neq \lambda$

2: $\wedge\ \langle x_{12} \rangle \circ q2.front_{12} = reverse\ (q1.back_0)$

3: $\wedge\ |q1.front_0| \leq 0$

4: $\Rightarrow\ q1.front_0 \circ reverse\ (q1.back_0) = \langle x_{12} \rangle \circ q2.front_{12} \circ reverse\ (\lambda)$

Fig. 6. A verification condition from the ensures clause of Enqueue

Figure 6 shows perhaps the most interesting of the VCs and is provided as an example of the simplicity of the proofs. Here the Greek letter lambda represents the empty string, and ◦ is the string concatenation operator. From 3 we can deduce that q1.front0 is empty. Given that, and the facts that the reverse of the empty string is itself the empty string we can apply the fact that the empty string is the identity for concatenation to simplify the required statement in 4 to that given in 3.

2.4 Argument of constant-time performance behavior

As noted in [15], we claim that this queue has amortized constant-time performance for all of its procedures. The reasoning for this is as follows: elements can only be added to the queue by the Enqueue operation which makes calls to Replica and Push. It is not hard to believe that Push is implemented in constant time, and for now let us assume that so is Replica (we will elaborate on this in a later section). The analysis of Dequeue has to be divided into two cases: If there is an element in the front queue then the performance argument is similar to that of Enqueue. However, if the front stack is empty the act of reversing the stack is a linear time event. This is why the claim is for amortized constant time, notice that for an element to be dequeued it has to be moved from the back to the front stack. This will happen only once, thus the cost of reversing a stack with **n** items is distributed around **n** calls to Dequeue. There

are ways to implement constant time reverse for stacks, however those require the use of cycles in their representation's references and that would prevent us from using the UVRT concept described in the next section as well as hampering the mechanisms that allow us to claim constant time replica of a stack.

This is not the first time that Copy on Write is used to maintain the illusion of having multiple copies of a mutable type when in reality there is only one. Some file systems have even implemented this to provide users with the impression that all users hold a unique copy of a mutable data-type even though the copies reside in the same place in the hard drive. Our technique to copying the stacks does not differ much from those with the exception that since the implementation in done at software's application level, the copying of objects can be finely tuned to match the needs of the data structure. Much more novel is the use of COW systems to efficiently implement immutable types.

3 A Shared Realization of Stacks with UVRT

This section explains a shared realization of Stacks with suitable annotations. In showing the correctness of this realization, the following key points need to be made:

1. There are no cycles in the representation.
2. There are no memory leaks.
3. Updating a stack with a push or a pop does not affect the values of any other object.
4. Stack replica is done in (amortized) constant time.
5. Reference counting and Copy on Write provide a mechanism to satisfy 3-4 given 1-2.

In our design, the first two points come "for free" because they are encoded in the specification of UVRT on which Stack implementation is based; this is the central purpose of the UVRT abstraction presented next. The third point is made in the discussion in Section 3.2. Due to space limitations, we give only an overview and allude to how points 4 and 5 are achieved.

A shared implementation of Stack_Template is based on the Ultimately Void Referencing Template, so we present its specification first.

3.1 Ultimately Void Referencing Template

Ultimately_Void_Referencing_Template (UVRT) is a specialized version of a pointer concept that is especially suitable for implementing of non-cyclic structures. Creating an instance of it allows a cycle-free structure to the actual type passed as an argument to the concept in instantiation. In the specification, 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. The concept has a shared state captured by Ref, a function that gives the "next" location for a given location and Contents, a function that gives the information value referenced by a given loca-

tion. There is a constraint on Ref, where the resulting set of applying the function Ref repeatedly to the set of Location until we reach the limit is just the set containing Void, and hence the name for the concept.

UVRT defines a programming type Pos to represent a pointer, mathematically modeled as a Location. Initially, each position takes the value Void.

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

    Defines Location: Set;
    Defines Void: Location;

    Var Ref: Location→Location;
    Var Content: Location→Info;
      Constraints Limit_Set_for(Location, {Ref},
                    Location) ⊆ {Void} which_entails
                                              Ref(Void) = Void;
      initialization ensures Ref[Location] = {Void} and
                   Info.Is_Initial[Content[Location]] = {True};

    Type Family Pos is modeled by Location;
      exemplar p;
      initialization ensures p = Void;
      Def var Accessible_Loc:℘(Location) = ({Void} ∪
                                  Closure_for(Location, {Ref},
                                Pos.Val_in[Pos.Receptacles]));
      finalization
        affects Ref, Content, Accessible_Loc;
        ensures Ref = λ q: Location.(
                  { if q ∈ {Void} ∪ Closure_for(Location,
        #Ref(q)     {#Ref}, #Pos.Val_in[
          Void         #Pos.Receptacle ~ {recp.p}])
                  otherwise
                                                          )
        and Content↑Accessible_Loc = #Content↑Accessible_Loc
        which_entails if #p ∈ {Void} ∪ Closure_for(Location,
               {Ref},#Pos.Val_in[#Pos.Receptacles ~ {recp.p}]),
        then Ref=#Ref and Accessible_Loc=#Accessible_Loc
                                        and Content=#Content;

    Constraints
      Info.Is_Initial[ Content[Location ~
                                 Accessible_Loc]] ⊆ {True} and
                   Ref[Location ~ Accessible_Loc] ⊆ {Void} and
                                          ||Accessible_Loc||: ℕ;
```

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

Oper Redirect_Ref_at(preserves p: Pos;
                     updates referent: Pos);
  affects Ref;
  requires p ∉ Closure_for(Location, {Ref}, {referent});
  ensures Ref = λ q: Location.( { #referent  if q=p
                                { #Ref(q)    otherwise )
                                    and referent = #Ref(p);

Oper Swap_Content_of(preserves p: Pos; updates I: Info);
  affects Content;
  requires p ≠ Void;
  ensures I = #Content(p) and
       Content = λ q: Location.( { #I        if q=p
                                 { #Content(q) otherwise );

Oper Relocate_to(preserves New_L: Pos; replaces p: Pos);
  affects  Ref, Accessible_Loc, Content;
  ensures p = New_L and Ref = λ q: Location.(
       { #Ref(q)   if q ∈ {Void} ∪ Closure_for(Location,
       { Void        {#Ref}, #Pos.Val_in[                ) and
                        #Pos.Receptacle ~ {recp.p}])
       { otherwise
      Content↑Accessible_Loc = #Content↑Accessible_Loc
  which_entails if #p ∈ {Void} ∪ Closure_for( Location,
           {Ref}, #Pos.Val_in[#Pos.Receptacles ~ {recp.p}]),
  then Ref=#Ref and Accessible_Loc=#Accessible_Loc and
                                          Content=#Content;

  (* Specification of operations: Follow_Ref, Are_Colocated,
    Is_Alost_Inaccessible, Is_Void, Set_to_Void omitted *)

end Ultimately_Void_Referencing_Template;
```

Fig. 7. A formal specification of Ultimately_Void_Referencing_Template

UVRT specifications have been designed with the goal of enabling automated verification, though we have not achieved this goal as yet. Specifically, through carefully-defined notations and theories, it avoids the use of quantifiers in assertions entirely. In understanding the constraint and the rest of the specifications, the following summary

of notations is useful. Formal definitions are given in function theory and its extensions.

- f↾S denotes the function restricted to S, a subset of f's domain.
- f[S] denotes the set of range values that correspond to S, a subset of f's domain.
- Closure_for(S, {f}, T) returns a set that results from applying f repeated to set the set of elements in T, a subset of S
- Limit_Set_for(S, {f}, T) returns a set that results from applying f to the limit of each member of the set T, a subset of S

The following notations are a part of the specification language that allow us to make assertions about all objects or a specific object of a certain type, because such assertions are necessary for specifying effects on shared representations.

- T.**Receptacles** 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

A key idea here is "accessibility" and it is specified by a variable mathematical definition Accessible_Loc. It is the set of reachable locations–locations produced by the Closure_for on all programming variables of a Pos type (i.e., all void-referencing pointer variables), unionized with Void. The finalization of a UVRT position ensures that p now is Void, all references other than p are not changed and the contents restricted to the set of accessible locations are the same. The *which_entails* clause states that if the location corresponding to a finalized p is accessible from a different reference, then Ref, Accessible_Loc and Content are not changed. This clause raises a proof obligation: specifically the assertion P which_entails Q leads to the obligation of proving Q, given P. Subsequently, Q becomes a useful lemma in the automated verification process.

The rest of the UVRT operations are discussed here briefly. Give_New_Loc allocates an unused location for a new Position; 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. Are_Colocated checks if two Positions point to the same memory location. Is_Almost_Inaccessible checks if p can be accessible from other Positions other than p. Is_Void checks if a Position is Void. Set_to_Void sets a Position to Void and finalizes all resources.

3.2 A (simpler) shared realization of stacks without constant-time replication

An implementation of stacks with an amortized constant-time replica operation is sufficiently complex that a full exposition of that code is not possible within the con-

straints of this paper. So we first discuss a shared realization of a stack interface with only Push, Pop, and Is_Empty operations. This implementation shares the global variables in the instantiation of UVRT, and verification needs to ensure the frame property that the code modifies only the representation of its parameter Stack and nothing else; specifically, all other stacks must remain unchanged. An interesting aspect here is that the frame property verification is just a part of the process along with the specifications of stack operations as explained here.

In this implementation, the Stack is represented using a UVRT pointer position, which will require the creation of an instance of UVRT with the appropriate realization in the library. For each of the memory displacements, the actual space required is simply the amount of memory displacement required by creating a new Position as defined by Ocpn_Disp_Incr inside Location_Referencing_Realiz. The *representation convention* states the set resulting from the Closure_for function with S intersected with the set resulting from the Closure_for function with all positions minus S is simply just the set containing Void. This indicates that all locations are independent and are not shared. The *correspondence* (i.e., the abstraction function) takes a Content function as well as the Ref function and returns the sequence of Entries. In the correspondence, the big PI denotes iterated concatenation of a series of strings, whereas the notation f^n denotes application of function f, n times. For brevity, only the code for Push is shown in the figure below.

```
Realization Simple_UVRT_Realiz for Stack_Template;
    uses Ultimately_Void_Refng_Template;

  Facility UVR_Fac is Ultimately_Void_Refng_Template(Entry)
    realized by Location_Referencing_Realiz;

  Type Stack = UVR_Fac.Pos;
    conventions ( Closure_for(Location,{Ref},{S}) ∩
                  Closure_for(Location, {Ref},
             Pos.Val_in[Pos.Receptacles ~ {recp.S}]) ⊆ {Void} );
    correspondence Conc.S =
```

$$\prod_{n=1}^{\|\text{Closure\_for(Location, \{Ref\}, \{S\})}\| \dot{-} 1} \left\langle \text{Content}(\text{Ref}^{n \dot{-} 1}(S)) \right\rangle;$$

```
  Procedure Push(updates S: Stack; clears E: Entry);
    affects Accessible_Loc, Content, Ref;
    ensures ( #Accessible_Loc ⊆  Accessible_Loc and
      Content1(Accessible_Loc ~ {recp.S})=
        #Content1(Accessible_Loc ~ {recp.S}) and
      Ref = λ q: Location.(#Ref(q)
               if  q  ∈ {Void} ∪ Closure_for (Location, {#Ref},
                        Pos.Val_in[Pos.Receptacle ~ {recp.S}]) )
      which_entails
```

```
        Stack.Val_in ↾(Stack.Receptacles~{recp.S}) =
                    Stack.#Val_in ↾(Stack.Receptacles~{recp.S});

    Var P: UVR_Fac.Pos;
    Give_New_Loc(P);
    Swap_Content_of(P, E);
    Redirect_Ref_at(P, S);
    Relocate_to(P, S);
  end Push;
              (* Code for other operations omitted *)
end Simple_UVRT_Realiz;
```

Fig. 8. An implementation of a Stack_Template using UVRT

The operation Push affects the set of accessible locations as well as the content and references in the accessible locations. Since Content, Ref, and Accessible_Loc are shared variables and the RESOLVE language is based on clean semantics [11] (meaning only the explicit parameter objects are allowed to be modified), the affects clause raises a proof obligation that only the parameters are modified and nothing else. Thus the "internal" ensures clause for Push documents how the internal shared variables are affected, the which_entails clause highlights subsequently that the effects are restricted to just the parameter stack. This obligation would not arise in the proof of the code for the Is_Empty procedure if it did not affect any shared state.

The internal ensures clause for Push states that the accessible locations prior to calling Push are contained within the new set of accessible locations, the contents of all other accessible locations other than the actual variable associated with S remain the same and all the references of q have not changed if they were originally in the set produced by the Closure_for operation above.

3.3 Outline of a shared realization of stacks including constant-time replication

The idea that a full deep copy of a structure with variable length is done in constant time should generate skepticism, and with good reason, since this is actually not possible. The trick for this is hiding from the clients that we do not really make a copy when Replica is called, and then doing the actual copy only when the values of the items replicated are about to diverge. This is technique is known as "Copy on Write" (COW) and a body of extensive work has studied it before, our implementation is based on [2]. This implementation relies on reference counting as its main mechanism. Whenever a stack is copied with Replica, the reference count of that stack is increased. As an object comes out of scope, its reference count is decreased.

The internal representation here takes the form (where Record is just a structure):

```
  Type Data_w_Count = Record
      Data: Entry;
      Ref_Count: Integer
```

```
end;

Facility UVR_Fac is
  Ultimately_Void_Refng_Template(Data_w_Count)
  realized by Location_Referencing_Realiz;
Type Stack = UVR_Fac.Pos;
```

The constant-time Replica procedure is straightforward and it is as shown below, here the variable Replica represents the value the function's return value:

```
Procedure Replica (restores S: Stack): Stack;
    Var Temp: Data_w_Count;
    Swap_Contents_of(S, Temp);
    Temp.Ref_Count := Temp.Ref_Count + 1;
    Swap_Contents_of(S, Temp);
    Relocate_to(S, Replica);
end Replica;
```

An Entry is copied only when Pop is called on a stack that has previously been replicated. The representation conventions differ from the simpler one in that it doesn't restrict the intersection of the positions holding stack representations to be Void. The correspondence, however, is similar. Again each procedure needs to ensure the frame property that when a Stack object is altered, none of the other stacks are modified. This is achieved by proving that whenever an object is modified, the number of references to its representation is one. Whenever a procedure wants to modify an object whose reference count is larger than one, the component proceeds to do a "deep copy" of the object's representation before doing any modifications. It is important to note that this deep copy is not a real deep copy, since only the top level object is created, and the values inside of it are "copied" by calling replica. Because of this we say that this action just "pushes" the copy one level down into the representation.

As explained in [2], there are plenty of potential pitfalls with reference counting. Given Resolve's design, all of them can be divided into two categories: Unmanaged aliasing or Cycles in the references. The first of the problems is impossible in the Resolve language. The second one is not a possibility in the UVRT, though it is possible in the more general Location Linking Template. Due to space restrictions we will not go into solutions to this problem, a potential solution is presented in [2].

4 Related Work

The general difficulties and challenges in verifying shared realizations is the topic of [3]. This earlier research focuses on the principles and is a useful starting point. However, it does not address shared realizations based on pointer behavior, automation issues, or verification with layered constructs.

The closure results necessary for proofs in this work, such as reachability, are established independently. This factoring out of reusable mathematical development

(independent of their application to the present verification problem) is a key reason for the simplicity of this treatment compared to, for example, [13]. Another key advantage of this method is that the logic used for the verification of the layered components and the pointer-based realizations is the same, there is no need for separate logics.

It would be interesting to study our approach to tree structures in relation to [18]. Our approach similarly involves establishing a mathematical theory of tree structures and using it to specify and reason about a Tree concept. However, the pointer-based implementation of the concept will be hidden (and verified once) using the UVRT templates described in this paper. Thus, such details will not routinely be raised in verification of client code.

The key principle of this research is to provide a way to reason about potential aliasing in the presence of references [17]. 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. [4][14][16]. 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 [6][7]. Separation Logic has emerged out of these ideas as the most promising in generating proofs involving references *within* a component [8][5].

The proposal for this research is to explore the possibility to specify arrays and pointers, thus be able to verify code involving pointer references. Kulczycki, et al presented the initial experimentation that involved the Location_Linking_Template. [12] The idea of this concept can have multiple implementations that provide additional functionality such as memory management and garbage collection.

5 Conclusions

Verification of shared realizations based on pointer behavior remains a challenge. This paper has provided an illustrative example to concretize the ideas involved in such verification. In the process, we have presented a layered implantation of a persistent Queue (one that behaves like an immutable one despite changes to its underlying representation). For each of the implementation's layers we provided an explanation of their features and annotations that can lead to its verification. We have shown that the layering allows us to write components that are relatively simple and amenable to modular verification, despite the rather complex nature of the entire composition. We have proved automatically the correctness of an immutable Queue based on the contract of a Stack.

We have introduced a restricted form of references in the UVRT that can be built on top of or more general LLT. The restricted nature of the UVRT allows us to construct Stacks that do not contain any cycles in their representation; this avoids the need for additional proofs. We have also introduced a list of properties we needed to prove to be able to create Stack representations that share parts of their representation with other stacks. We have introduced a Copy on Write methodology for preserving

sound invariants across the shared representations, and we have used the constraints on UVRT to discharge one of the main requirements for the COW system.

Automated verification of shared realizations is not possible as of yet, but we are making progress on this front. Future work will provide automated verification of components based on the UVRT as well as their performance profiles.

Acknowledgments. We thank members of the RESOLVE/Reusable Software Research Groups at Clemson and Ohio State for their inputs at various points in this research. Our special thanks are due to Joan Krone, Joe Hollingsworth, Bill Ogden, and Bruce Weide. This research is funded in part by U. S. National Science Foundation grants CCR-0113181, DMS-0701187, and CCF-1161916.

References

1. Sitaraman M., et al., "Building a Push-Button RESOLVE Verifier: Progress and Challenges," Formal Aspects of Computing 23 (3), Springer, 2011, 607-626.
2. Adcock, B. "Working towards the verified software process", Ph.D. Thesis, Ohio State University, 2010.
3. Ernst, et al., "Modular Verification of Data Abstractions with Shared Realizations". IEEE Transactions on Software Engineering (TSE), Volume 20, Number 4, April 1994, 288-307
4. 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.
5. Gardner, P., and Zarfaty, U. An introduction to context logic. Proc. Logic, Language, Information and Computation, LNCS 4576, Springer, 2007, 189-202.
6. Hatcliff, J., et al., Behavioral Interface Specification Languages. Dept. of EECS, University of Central Florida, CS-TR-09-01, March 2009
7. 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.
8. Kassios, I. Dynamic frames: support for framing, dependencies and sharing without restrictions. Proc. FM 2006: Formal Methods, LNCS 4085, Springer, 2006, 268-283.
9. Kirschenbaum, J., et al., Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping? Proceedings 11th International Conference on Software Reuse, Springer LNCS 5791, September 2009, 31-40.
10. Krone, J., Sitaraman, M., and Ogden, W.F., "Performance Analysis Based Upon Complete Profiles", Proceedings of SAVCBS 2006: FSE Workshop on Specification and Verification of Component-Based Systems, Portland, OR, Nov. 2006, pp. 3-10.
11. Kulczycki, G. "Direct Reasoning". Clemson University, 2004
12. Kulczycki, G., et al., The Location Linking Concept: A Basis for Verification of Code Using Pointers. VSTTE'12 Proceedings of the 4th international conference on Verified Software: theories, tools, experiments, 34-49
13. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 171–182.
14. 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.
15. Burton, F. W. An efficient functional implementation of FIFO queues. Information Processing Letters, 14(5):205–206, July 1982.

16. O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. Proceedings 15th International Workshop on Computer Science Logic (CSL 2001), LNCS 2142, Springer, 2001, 1-19.
17. Weide, B. W., and Heym, W.D. Specification and verification with references. Proc. OOPSLA Workshop on Specification and Verification of Component-Based Systems, 2001.
18. Wies, T., Muñiz, M., Kuncak, V.: An Efficient Decision Procedure for Imperative Tree Data Structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476–491. Springer, Heidelberg (2011)