The RESOLVE Approach for Achieving Modular Verification: Progress and Challenges in Addressing Aliasing

Yu-Shan Sun, Alan Weide, Diego Zaccai, Paolo A. G. Sivilotti, and Murali Sitaraman

Technical Report RSRG-17-03
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

October 2017

Copyright © 2017 by the authors. All rights reserved.
The RESOLVE: Approach for Achieving Modular Verification: Progress and Challenges in Addressing Aliasing

Yu-Shan Sun\textsuperscript{2}, Alan Weide\textsuperscript{1}, Diego Zaccai\textsuperscript{1}, Paolo A. G. Sivilotti\textsuperscript{1}, and Murali Sitaraman\textsuperscript{2}

\textsuperscript{1} The Ohio State University, Columbus OH 43221, USA
weide.3@osu.edu, zaccai@osu.edu, paolo@cse.ohio-state.edu
\textsuperscript{2} Clemson University, Clemson SC 29634, USA
yushans@clemson.edu, murali@clemson.edu

Abstract. The overall research objective of the RESOLVE project is to enable automated verification of sequential and parallel object-based programs in a modular fashion. Aliasing presents a challenge because it complicates automated, modular verification. The paper presents a three-pronged approach to addressing this challenge, including progress towards (i) automated verification that leverages a promising notion of clean semantics in which objects in client code are decoupled by language and software design; (ii) verification for data abstraction implementations involving explicit (and unavoidable) aliasing using a concept that captures acyclic reference behavior; and (iii) specification of conditional effects clauses for verifying parallel execution of operations on a shared object without violating abstraction.

1 Introduction

Non-trivial programs designed with software engineering consideration will be invariably composed from reusable components—often components that encapsulate data abstractions. Prior work by others and us has established that it is possible to specify and verify component-based sequential software for full functional behavior (not just a subset of properties), automatically, one component at a time, in a modular fashion using design-by-contract. However, automated tools will require software developers to understand and write specifications and code annotations, though they may not have to be involved in proving [24]. RESOLVE language and tools are designed to facilitate specification, implementation, and modular verification of component-based software [44][43], and the ideas have been adopted to popular languages, including Java [50].

A key complication in reasoning is the presence of object references and aliasing [9][21][31][48]. In the context of using separation logic to tackle cross-boundary aliasing among components, Peter O’Hearn (one of the inventors of separation logic) and his coauthors conclude [16]: “Fundamentally, because it concerns aliased pointers, freedom from interference is extremely difficult to
protect against using programming language restrictions, and too expensive to
protect against with runtime checking. It is better to say that if there is no
interference then refinement reasoning is sound, rather than to say that it is
unconditionally sound.” Not surprisingly, aliasing and sharing are central obsta-
cles in verifying parallel programs manipulating objects, as well [32]. Other most
relevant related work is summarized in Section 2.

Though we do not use separation logic, the goal of the first component of our
research is roughly in line with O’Hearn’s quote above. At the core of the solution
to the aliasing problem is a notion of clean operation calls whereby effects of calls
are restricted to objects that are explicit parameters or to global objects that
are explicitly specified as affected [25]. Under this notion, regardless of the level
of granularity, syntactically independent operation calls are always unentangled.
The solution relies on suitable language and software design and it is exemplified
through automated verification of a simple, but illustrative example of a search
operation on a Queue data abstraction in Section 3.

To handle verification in the presence of explicit use of references (and any
unavoidable aliasing), ideally hidden in data abstraction implementations, we
specify and use a novel concept\footnote{An earlier version of this concept appears in\cite{27}.} that captures acyclic reference behavior and is
suitable for implementation of data abstractions, such as queues, list, or trees.
While the concept cannot be presented and discussed here due to space con-
straints, we give a summary of the approach detailed in [45] in Section 4.

Section 5 discusses safe parallel execution of code such as the search oper-
ation (that is naturally disentangled) and code that involves parallel operation
calls to manipulate a shared object. It outlines an approach that involves inter-
ference contracts [47] and explains the novel idea of conditional effects clauses.
These clauses are layered on assertions for behavioral specifications, making it
apparent that verification of safe parallel execution is also layered and dependent
on verification of behavioral correctness.

The last section contains a summary of ongoing research and directions for
further research.

2 Related Work

One central problem for verification concerns objects and aliasing [29]. Separa-
tion logic, one well-known approach for addressing the problem, is an extension
of Hoare’s logical rules to address this challenge and properties about the heap
\cite{39}. Examples of verification using separation logic in Coq \cite{13} and in VeriFast
to verify Java and C programs \cite{22}. Automating verification with separation logic
is the topic of \cite{5,7,8,35,36}. There have been attempts to combine modularization
and address information hiding in conjunction with separation logic \cite{33,34},
though problems remain in generalizing the approach to encompass many object
instances.

Dynamic frames are designed to addresses the frame problem for shared
and encapsulated references\cite{23}. In dynamic frames the idea of an infinite set
of locations, \textit{Loc}, with each subset being a “region” is introduced. In dynamic frames, a \textit{footprint} is the set of fields that a method can modify: A \textit{reads} mode indicates that a region is only being read by an operation, unlike a \textit{modifies} mode. The idea of dynamic frames has been used in multiple frameworks for specification and verification. JML* [49] is a dynamic frame extension of Java Modeling Language (JML), a specification language built on top of Java [28]. Dafny and KeY use dynamic frames in their reference specifications [10][30][42]. KeY uses dynamic logic to generate fully automated proofs [4], but includes features for interactive proofs as well.

Region logic is similar to dynamic frames where the specification defines a region and global states [3]. However, region logic formulates the specifications using only first-order logic to facilitate automation [2]. Decision procedures when using region logic are discussed in [40][41]. A region expression $G$ of type \textit{rgn} is used to define a region in the heap. Similar to dynamic frames, the region being \textit{modified} is represented using the keyword \textit{wr} or write effects and a region that is simply read is represented using the keyword \textit{rd}. These notions form the basis for using regions to specify parallel programs in [6].

There is significant overlap on handling aliasing and concurrency within a common framework between the RESOLVE approach and the research direction of Kappa [11][12]. Whereas the RESOLVE approach has defined and used as its basis a language with clean semantics and built on a system for verification of full functional behavior using abstract specifications in a sequential setting, Kappa is significantly more advanced in its treatment of concurrency. The treatment of aliasing through ownership in Kappa and the use of a Java-like language are other key distinctions.

Since race conditions pose such a significant challenge in the development of correct concurrent software, their detection and elimination has been the subject of considerable research. On one hand, static techniques have been developed to limit the inadvertent introduction of races [1][15][18][19]. These techniques are often cast as extensions of typing, decoupling the problem from full functional verification. On the other hand, such techniques are necessarily imprecise, giving rise to significant work in dynamic techniques which improve precision at the expense of execution overhead [17][37][38]. These two approaches have complementary strengths—or rather weaknesses—making their combination a natural way to explore various trade offs.

### 3 Verification Leveraging Clean Semantics

The RESOLVE approach relies on hiding explicit references and aliasing where possible (e.g., see [20]), through abstraction, to simplify reasoning as illustrated in this section. Where unavoidable, it captures locations and links between locations using an interface specification to facilitate reasoning about aliasing as summarized in Section 4.

An interface specification (concept): \texttt{Queue.Template} is parameterized by a generic type \texttt{Item} and a \texttt{MaxLength} integer value that indicates the bound. The
complete specification with a description is given in the Appendix. This specification imports string theory notations to define a type \texttt{Queue} that is modeled abstractly as a mathematical string (or a sequence) of abstract \texttt{Item} values. No references are involved in this specification. The type definition is shown in Listing 1. Using an example \texttt{Queue}: \texttt{q}, it specifies that every \texttt{Queue} is initially empty and that the length of \texttt{q} must be strictly less than or equal to the bound.

\begin{verbatim}
Type Queue is modeled by Str(Item);
  exemplar q;
  constraints |q| <= Max_Length;
  initialization ensures q = empty_string;

Listing 1. Mathematical Model for Queue

The concept defines operations such as \texttt{Enqueue}, \texttt{Dequeue}, \texttt{SwapFirstEntry} and \texttt{Length}. These operations have been designed and specified to avoid aliasing that arises when queues contain non-trivial objects [20] and to facilitate clean semantics. In addition to the operations, a swap operator is defined on all types to facilitate data exchange without deep or shallow copying [20].

We can create extensions (enhancements) for \texttt{Queue_Template} to define additional operations such as \texttt{Split}, \texttt{Is_Present} and \texttt{Append}. The descriptions for these enhancements can be found in the Appendix. In the figure below, we have instantiated a \texttt{Queue} of \texttt{Integers} with these enhancements to add a parallelizable version of \texttt{Is_Present}. Parallel\texttt{Is_Present}'s ensures clause uses a mathematical predicate: \texttt{Is_Substring(<x>, q)}, which evaluates if the singleton string containing \texttt{x} is in \texttt{q} (recall that \texttt{Queues} are mathematically modeled as a string).

\begin{verbatim}
... Operation Split(updates P: Queue; replaces Q: Queue);
... Operation Is_Present(restores E: Entry;
...   restores Q: Queue): Boolean;
... Operation Append(updates P: Queue; clears Q: Queue);
... requires |P| + |Q| <= Max_Length;
... ensures E = Is_Present(x) && E = Is_Present(y);

Fig. 1. Automated Verification of Parallel\texttt{Is_Present}

The code in Figure 1 splits the incoming \texttt{Queue} into two different parts. The front section of the \texttt{Queue} is stored in \texttt{r}, while the rest remain in \texttt{q}. It then
The RESOLVE Approach for Achieving Modular Verification

makes two separate calls to \texttt{Is\_Present} to check if \(x\) is in the modified \(q\) and if it is in \(r\). As long it is in one of them, \texttt{Parallel\_Is\_Present} returns \textit{true}, otherwise it is \textit{false}. At the end of the procedure, we put the contents back together into \(r\) by calling the \texttt{Append} operation. The last statement in the code makes use of the swap operator that introduces no aliasing and is also efficient [20]. This implementation can be verified to be correct in a modular fashion by the existing RESOLVE verifying compiler for which a web interface is available [14]. Verification of \texttt{Parallel\_Is\_Present} is modular and it relies only on the specifications of reused operations: \texttt{Split}, \texttt{Is\_Present}, and \texttt{Append} and not on their actual implementations. The guarantee that \(x\) is preserved, i.e., that it is never changed during the execution of the operation, is attained by ensuring that it is used only in places where it will be preserved. For example, \(x\) is passed as a parameter to the call \texttt{Is\_Present} and the code will be deemed syntactically incorrect, if \texttt{Is\_Present} merely restored \(x\) (which would allow it to modify it temporarily.)

Work is ongoing to extend the verifying compiler for parallel software. The sequential implementation is parallelizable by placing the two calls to \texttt{Is\_Present} in the \texttt{Cobegin} construct. However, it is safe only if we can assume a language with clean semantics, because otherwise \(q\) and \(r\) could be aliased. In addition to absence of aliasing, the parallelization also demands that the shared object \(x\) is preserved. The parallelization below is safe and non-interfering and this is guaranteed syntactically.

\begin{verbatim}
Cobegin
    ans1 := Is\_Present(x, q);
    ans2 := Is\_Present(x, r);
end;
\end{verbatim}

\textbf{Listing 2.} Code that is Parallelizable

4 Verification with Explicit References Summary

The approach we have employed in prior and ongoing work to reason about sequential software where references and aliasing are needed explicitly (e.g., a linked structure implementation of queues, lists, or trees) is to use a “sharing” concept specification, that is similar to other data abstraction specifications, but with shared global state [25][26][27][45]. Such a concept provides a reference type that is modeled mathematically as a set of \texttt{Locations} (an abstraction of addresses) and operations to allocate storage, change the value of item pointed to by a reference, alias references, check if two references are aliased, follow a reference to (a) next one, check if a reference is void (or \texttt{null}), and so on.

For purposes of simplifying this discussion, it is sufficient to envision the shared state as two mathematical functions \texttt{Ref} and \texttt{Content}, that are functions from \texttt{Locations} to (next) \texttt{Locations} and \texttt{Locations} to the item pointed by the reference. The detailed specifications employ standard, higher-order logic and software verification reuses results from extensions to function theory.
Cobegin
    Boolean b1 := Is_Void(p);
    Boolean b2 := Is_Void(q);
end;

Listing 3. A Parallelizable Code with Explicit References

Clearly some of the operations affect the global states Ref and Content, and some do not. For a simple example, safe execution of Listing 3 needs no proof (because Is_Void preserves its reference and is oblivious to global state) though p and q may be aliased. However, Listing 4 makes a call to Follow_Ref that updates p, therefore to ensure safe execution it needs to proof that p \neq q.

Cobegin
    Follow_Ref(p);
    Boolean b2 := Is_Void(q);
end;

Listing 4. Another Parallelizable Code with Explicit References

While not an issue for any of the examples presented in this paper, the aliasing problem arising from passing repeated arguments to a call is an important side issue and a specification-based solution approach is outlined in [25].

5 Use of conditional effects clauses in interference contracts

Related work has focused primarily on confirming the non-interference of concurrently executing operations through annotating the maximal effects operations might have on their parameters [6][46]. In some cases, these effects have been defined with respect to distinct regions of memory, and in others with respect to objects themselves. In all cases they have been unconditional, that is, the effects of a method are the same no matter the values of the objects in question. There are numerous benefits to this approach. Notably, it reduces non-interference proofs to a purely syntactic check, which requires neither sophisticated static analysis nor the ability to reason about the functional behavior of programs. However, in the context of RESOLVE we can reason about the functional behavior of a program and introduce conditional effects clauses, which enumerate the potential effects of an operation in terms of a partitioning of the concrete state space of its parameters⁴ and conditional on the incoming abstract values of its parameters. We define the property of non-interference as follows: the statements within a Cobegin statement are non-interfering if and only if every affects-mode partition in one operation call in a Cobegin statement is oblivious-mode in each other one.

⁴ As detailed in [47], a partition may be in one of three modes: affects, preserves, or oblivious.
Interference contract Sample detangles Queue_Template;

Partition for Queue is (a, b);

Operation Enqueue (clears e: Item, updates q: Queue);
    affects q@b;
    when |q| = 0 affects q@a;
    OTHERWISE (|q| > 0) oblivious to q@a

Operation Dequeue (replaces r: Item, updates q: Queue);
    affects q@a;
    -- oblivious to q@b

Operation SwapFirstEntry (updates e: Item, updates q: Queue);
    affects q@a;
    -- oblivious to q@b

end interference contract Sample

Listing 5. A sample interference contract which detangles Queue_Template

Effects clauses are written in an interference contract, which is an extension of an abstract concept specification for expressing the effects of operations on their parameters. A sample interference contract for a class of queue implementations is given in Listing 5. The operation Enqueue is annotated with conditional effects clauses which, when combined with the effects clauses for the Dequeue operation, imply that whenever the incoming Queue has at least one element in it (i.e., |q| > 0), each partition which is affected by Enqueue (i.e., partition q@b) is an oblivious-mode partition in Dequeue, and therefore the statement

Cobegin
    Enqueue(x, q);
    Dequeue(y, q);
end;

is non-interfering. Furthermore, because Dequeue requires that |q| > 0, in any situation where either Dequeue and Enqueue could be called, they can be called in parallel with each other. Two items of note here. The clean semantics of RESOLVE ensures that x and y are not aliased. Verification of non-interference relies on conditional effects which in turn rely on abstract values, and hence, it needs to be preceded by and layered upon verification of behavioral correctness assertions discussed in Section 3.

In [47], we discuss in detail how multiple interference contracts can be defined for a single concept, each of which express different levels of parallelism. That paper introduces three different interference contracts for the Queue concept described here, each of which support the safe concurrent execution of different classes of operations. Table 1 provides a summary of which operations may be
executed concurrently with \texttt{Enqueue} in each of those interference contracts. The interference contracts themselves are omitted from this paper for brevity.

Table 1. Operations on \texttt{Queue\_Template} which can be parallel-composed with \texttt{Enqueue} are marked with ✓. \texttt{Dequeue\_From\_Long} is an auxiliary operation that requires $|q| > 1$.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
Interference Contract & #1 & #2 & #3 (Sample) \\
\hline
Dequeue & ✓ & & ✓ \\
Dequeue\_From\_Long & & ✓ & ✓ \\
Swap\_First\_Entry & ✓ & ✓ & ✓ \\
\hline
\end{tabular}
\end{table}

Each of these interference contracts is respected by at least one realization (implementation). The proof that a realization actually respects its interference contract can be carried out by inspecting the minimal modification mode of a partition at a given state of the implementation. Partitions may have one of three modification modes: oblivious, preserves, and affects. These three modes are ordered by $\succeq$ (read “is above” or “is more restrictive than”), with oblivious $\succeq$ preserves $\succeq$ affects.\(^5\) An implementation respects its interference contract if the mode of a partition in each constituent statement of an implementation is related by $\succeq$ to the mode of that partition defined in the interference contract.

6 Summary and Future Directions

This paper has summarized the ongoing RESOLVE approach for achieving modular verification of component-based software. Through language and software design with abstractions, explicit reference behavior is avoided where possible. This approach makes it easier to reason about routine sequential (and parallel) software, and a verifying compiler has been built and used to verify components in software engineering courses. For handling unavoidable references and aliasing, such as in the implementation of linked data structures, a modular approach that relies on specifying and using an interface that abstracts references as locations and global functions to capture the behavior of operations in a linked system of references is used. At present, while the compiler generates verification conditions for correctness for linked data structures, automation in proving them is a work in progress. Definition of a formal system of proof rules that leverages abstraction and interference contracts for safe parallel execution on a shared object is in progress as a first step toward extending the compiler.

\(^5\) Note that no proof is discharged to confirm a partition is affects-mode. A partition is always at least affects-mode.
Acknowledgments

We thank several members of our research groups at Clemson and Ohio State who have contributed to the discussions on topics contained in this paper. This research is funded in part by US NSF 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.

Appendix

The listing below contains the specifications for Queue and its associated operations. It is parameterized by a generic type Item and a Max_Length integer value for the bound. The evaluates mode allows us to pass in an Integer expression during instantiation. Notice that it requires that the Max_Length to be greater than 0 and imports the definitions defined in String Theory.

Concept Queue_Template (type Item; evaluates Max_Length: Integer);
  requires Max_Length > 0;
  uses String Theory;

Type Queue is modeled by Str(Item);
  exemplar q;
  constraints |q| <= Max_Length;
  initialization ensures q = empty_string;

Operation Enqueue (alters e: Item, updates q: Queue);
  requires |q| < Max_Length;
  ensures q = #q o <#e>;

Operation Dequeue (replaces e: Item, updates q: Queue);
  requires q /= empty_string;
  ensures #q = #e o q;

Operation SwapFirstEntry (updates e: Item, updates q: Queue);
  requires q /= empty_string;
  ensures <e> = substring(#q, 0, 1) and
    q = <#e> o substring(#q, 1, |#q|);

Operation Length (restores q: Queue) : Integer;
  ensures Length = |q|;
end Queue_Template;

Listing 6. Queue_Template

The type Queue is modeled as a string of Item and using an example Queue: q, it states all Queue’s length cannot exceed Max_Length and that all Queue’s are initially empty (or empty_string).
The pre-condition (requires clause) and post-condition (ensures clause) are strictly mathematical and are stated for each of the operations. In addition to these clause, each operation provides specification parameter modes, which explicitly states the effect of the operation on the parameter on exit. The alters mode in Enqueue allows e to pass a meaningful value, but does not specify the outgoing value of e. In contrast, the updates mode passes a meaningful value and its outgoing value will be updated according to the ensures clause. The |...| function used in the requires clause returns the length of q and is used to specify that the queue is not full. The # notation indicates incoming value, o is the concatenation operator and <...> is a function that converts an entry to a singleton-string, therefore Enqueue ensures that the outgoing q is equal to #q concatenated with a singleton-string #e.

The Dequeue operation uses the replaces mode, where it takes an arbitrary passed in value and modifies it to give e a meaningful value at the end of the operation. In order to specify the ensures clause of SwapFirstEntry, it uses the substring function to return a substring of #q. Lastly, the Length operation uses the restores mode to indicate that the q = #q, but its value could have been temporary modified by Length.

Listing 7 provides two enhancement specifications for Queue_Template that contain additional operations which can be implemented using a combination of Queue primary operations. Is_Present uses the preserves mode, where the operation cannot modify e at all and ensures that it returns true if <e> is a substring of q. The Append_to requires the sum of the p's and q's length is less than Max_Length and ensures that p is equal to #p concatenated with #q. Additionally, the clear mode specifies that the outgoing q is equal to the empty_string. Lastly, the Split operation ensures that #p is equal to q concatenated by p, meaning that some of the contents in #p have been transferred over to q.

Enhancement Search_Capability for Queue_Template:

Operation Is_Present(preserves e: Item;
                      restores q: Queue): Boolean;
    ensures Is_Present = Is_Substring(<e>, q);
end Search_Capability;

Enhancement Append_Split_Capabilities for Queue_Template:

Operation Append_to(updates p: Queue; clears q: Queue);
    requires |p| + |q| <= Max_Length;
    ensures p = #p o #q;

Operation Split(updates p: Queue; replaces q: Queue);
    ensures #p = q o p;
end Append_Split_Capabilities;

Listing 7. Extensions of Queue_Template
References

The RESOLVE Approach for Achieving Modular Verification


