Modular Verification in the Presence of Shared Objects and Concurrency

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

Technical Report RSRG-19-02
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

January 2019

Modular Verification in the Presence of Shared Objects and Concurrency

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

Alan Weide
Computer Science & Engineering
Ohio State University
Columbus OH, USA
weide.3@osu.edu

Diego Zaccai
Computer Science & Engineering
Ohio State University
Columbus OH, USA
zaccai.1@osu.edu

Paolo A. G. Sivilotti
Computer Science & Engineering
Ohio State University
Columbus OH, USA
paolo@cse.ohio-state.edu

Murali Sitaraman
School of Computing
Clemson University
Clemson SC, USA
murali@clemson.edu

***Abstract*—Aliasing presents a challenge because it complicates automated, modular verification. This paper presents a three-pronged approach to addressing this challenge: (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. This approach is presented in the context of the RESOLVE project, which aims to enable automated verification of sequential and parallel object-based programs.**

I. Introduction

Non-trivial programs designed with software engineering consideration will be invariably composed from reusable components—often components that encapsulate data abstractions. Prior work has established that it is possible to specify and automatically verify component-based sequential software for full functional correctness in a modular fashion using design-by-contract. While software developers do need to understand and write specifications for such verification to proceed, the proofs may be fully automated [1]. The RESOLVE language and tools are designed to facilitate specification, implementation, and fully automatic modular verification of component-based software [2], [3], and the ideas have been adopted to popular languages, including Java [4]. A key complication is reasoning in the presence of aliasing of object references [5]–[8], which is even harder when concurrency is involved [9], [10]. O'Hearn *et al.* note: "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." [11]

This research is funded in part by NSF grants CCF-1161916 and DUE-1022941. Opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

The ideas in this paper are mostly about realizing O'Hearn's guidance and showing that "if there is no interference then refinement reasoning is sound" for code with *fork-join parallelism*. The approach eases automation by showing "there is no interference" for most software through language and software design. 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 [12]. Under this notion, regardless of the level of granularity, syntactically independent operation calls are always unentangled. The solution approach is exemplified through a simple, but illustrative example of a search operation on a Queue data abstraction in Section III. The paper summarizes how the solution can be generalized to ease client-side reasoning by conceptualizing and developing suitable interfaces for shared objects in Section IV, including when they are manipulated in parallel in Section V, and for subsequently ensuring that the implementations are correct refinements, i.e., that they are actually non-interfering in Section VI. The ideas are detailed in three Ph. D. dissertations (two of which are published [4], [13]) and a Masters thesis [14].

II. Related Work

Some kind of frame property (*i.e.*, objects not mentioned do not change) is fundamental to modular verification. In languages with ubiquitous references, such as Java, potential aliasing abounds and complicates the establishment of such a property. Separation logic, one well-known approach for addressing this complication, is an extension of Hoare's logical rules to include heap properties [15]. Separation logic has been used for verification in Coq [16] and in VeriFast [17]. Automating verification with separation logic is the topic of [18]–[21]. Although there have been attempts to combine

modularization and address information hiding in conjunction with separation logic [22], [23], 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 [24]. Region logic is similar to dynamic frames where the specification defines a region and global states [25]. However, region logic formulates the specifications using only first-order logic to facilitate automation [26].

The RESOLVE solution approach is fundamentally different in that the need to reason about references is exceedingly rare. The normal case, by language and component design, is that aliases are entirely absent. The implications of applying these ideas to a Java-like context are presented in [4].

III. Verification Leveraging Clean Semantics

The RESOLVE approach relies on hiding, through abstraction, explicit references and aliasing where possible (e.g., see [27]) to simplify reasoning. Experience has shown that this hiding is possible even in cases where aliasing appears inevitable. This section illustrates how aliases can be hidden when an object is shared between concurrent threads, and the next section describes reference hiding when objects share communal representations.

Consider an interface specification (*concept*) `Queue_Template`, parameterized by a generic type `Item` and a `Max_Length` integer value. This concept defines a type `Queue`, which is modeled as a mathematical string (*i.e.* a sequence) of abstract `Item` values. No references are involved in this type definition. Listing 1 shows this definition, including the abstract invariant and the initial state of a new `Queue`.

```
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 also defines operations such as `Enqueue`, `Dequeue`, `SwapFirstEntry` and `Length`. These operations have been designed and specified to avoid aliasing that arises when queues contain non-trivial objects. For example, a client does not retain a reference to an item it enqueues. In addition to these operations, a swap operator is defined on all types to facilitate efficient data exchange without deep or shallow copying [27].

Extensions (*enhancements*) for `Queue_Template` specify additional operations such as `Split`, `Is_Present` and `Append`. Figure 1 shows the instantiation of a `Queue` of `Integers` with these enhancements to add a parallelizable version of `Is_Present`. `Parallel_Is_Present`'s ensures clause uses a mathematical predicate: `Is_Substring(<x>, q)`, which evaluates if the singleton string containing `x` is in `q` (recall that `Queues` are mathematically modeled as a string).

The code in Figure 1 splits the incoming `Queue` into two parts: The front is stored in `r`, while the rest remains in `q`

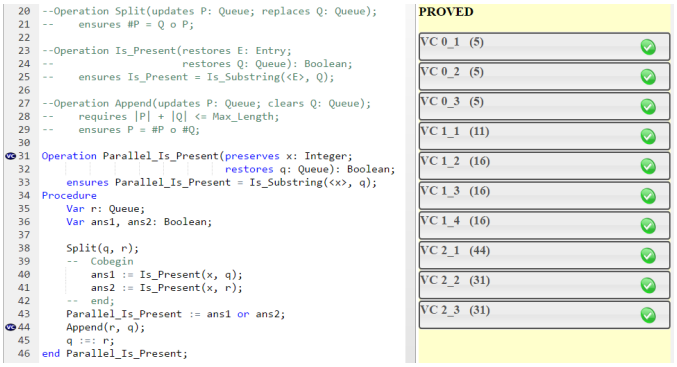

Fig. 1. Automated Verification of `Parallel_Is_Present`

. It then makes two separate calls to `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, `Parallel_Is_Present` returns *true*, otherwise it is *false*. At the end of the procedure, we put the contents back together into `r` by calling the `Append` operation. The last statement in the code makes use of the swap operator, which efficiently moves data from `r` to `q` without aliasing. This implementation is verified by the RESOLVE verifying compiler, for which a web interface is available [28]. Verification of `Parallel_Is_Present` is entirely modular, relying only on the specifications of `Split`, `Is_Present`, and `Append` and not on their actual implementations.

The guarantee that `x` is preserved, *i.e.*, never changes during execution of the operation, is attained by ensuring that it is used only in places where it is preserved. For example, `x` is passed as a parameter to the call `Is_Present` and the code will be deemed syntactically incorrect, if `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 parallel calls to `Is_Present` are safe only in a language with clean semantics, because otherwise the contents of `q` and `r` could be aliased. In addition to the absence of aliasing, the parallelization also demands that the shared object `x` be preserved.

IV. Client-Side Verification with Shared Objects

One central idea is to enlarge the set of programs where non-interference reasoning in client code is simplified, even when underneath the cover of an interface, multiple queue objects may live within a shared or communal space. Clients rely on the guarantee that it is safe to compute on these objects in parallel, while separately we ensure that this guarantee is sound, i.e., that operations on one object do not affect other objects in that communal space.

We have demonstrated the feasibility of designing interfaces of communal concepts, such as "splittable arrays" for working with arrays [29]. We have also developed a formal specification of a communal "exploration tree" concept [14] with a shared space bound for trees from an instance. The only dependency among trees seen in the specification is that addition of new nodes to one tree might affect how many can be added to another. Beyond that the interface permits navigating and modifying trees, and extracting their subtrees that all live in a communal space as though they are

independent thus enabling parallel manipulation. Without a careful interface design of the communal concept, establishing the independence of trees and their (decomposed) subtrees requires a non-trivial proof or an intricate use of region logic by programmers [30].

V. Use of conditional effects clauses in non-interference contracts

Reasoning about the independence of concurrent threads in client code requires exposing more information than what is necessary for functional behavior specification. This information is exposed via a *non-interference contract*, as illustrated in Listing 2 [31]. The contract introduces names for the segments into which a component is partitioned (a and b in this case). These segments are independent of implementation particulars. The non-interference contact also describes the effect of each operation on these segments in terms of three possible modes: *affects*, *preserves*, and *ignores* (the default). The simplest proof for parallel blocks is that two operations, *op1* and *op2*, are said to be *non-interfering* when, for all segments s, $s \in \text{affects}(op1) \Leftrightarrow s \in \text{ignores}(op2)$.

Work by others has focused primarily on confirming the non-interference of concurrently executing operations through annotating the maximal effects operations might have on their parameters [30], [32]. 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*, conditional on the incoming *abstract values* of variables.

```
noninterference contract LookupOffset for
    BoundedQueueTemplate;
  partition for Queue is (head, tail, offset);

  operation Enqueue (clears e: Item, updates q:
    Queue);
    affects q@tail;
    preserves q@offset;
    when q = empty_string affects q@head;

  operation Dequeue (clears e: Item, updates q:
    Queue);
    affects q@head, q@tail, q@offset;
    restructures q@head, q@tail;

  operation SwapFirstEntry (updates e: Item,
    updates q: Queue);
    affects q@head;
    preserves q@offset;

end LookupOffset;
```

Listing 2. A sample interference contract which detangles `Queue_Template` and includes conditional effects clauses.

In this case, 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., $|\texttt{q}| > 0$), each partition which is *affected* by `Enqueue` (i.e., partition `q@b`) is *ignored* by `Dequeue`, and therefore the statement

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

is non-interfering. Furthermore, because `Dequeue` requires that $|\texttt{q}| > 0$, in any situation where either `Dequeue` and `Enqueue` *could* be called, they can be called in parallel with each other. There are 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 III.

VI. Verification with Explicit References

Ultimately not all reasoning about aliasing interference can be avoided. For this reasoning, while separation logic can be used, the approach we have employed (e.g., a linked structure implementation of queues, lists, or trees) is to use a "sharing" concept specification [12], [13], [33] that provides a reference type that is modeled mathematically as a set of `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 `null`), and so on. The discussion in [12] also contains a solution to avoid the classical parameter aliasing problem on calls with repeated arguments, a solution that can be realized in C++ with move semantics.

The detailed specifications employ standard, higher-order logic and automated verification reuses results from extensions to function theory [13]. A key contribution of [13] is in noting when abstraction functions in explicitly reference-based realizations can be regionalized to simplify reasoning about non-interference, meaning that the general complexity can be avoided for most "layered" realizations of shared concepts.

VII. 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 (including in upper division software engineering courses). For handling unavoidable references and aliasing, such as in the implementation of linked data structures, a modular approach is used. This approach 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. 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.

REFERENCES

[1] V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß, "The 1st Verified Software Competition: Experience Report," in *FM 2011: Formal Methods*, ser. Lecture Notes in Computer Science, M. Butler and W. Schulte, Eds. Springer Berlin Heidelberg, 2011, vol. 6664, pp. 154–168.

[2] M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. M. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. W. Weide, "Building a Push-Button RESOLVE Verifier: Progress and Challenges," *Formal Aspects of Computing*, vol. 23, no. 5, pp. 607–626, 2011.

[3] M. Sitaraman and B. Weide, "Component-based Software Using RESOLVE," *SIGSOFT Software Engineering Notes*, vol. 19, no. 4, pp. 21–22, Oct. 1994.

[4] D. S. Zaccai, "A Balanced Verification Effort for the Java Language," PhD Thesis, The Ohio State University, 2016.

[5] C. Boyapati, B. Liskov, and L. Shrira, "Ownership Types for Object Encapsulation," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03. New York, NY, USA: ACM, 2003, pp. 213–223.

[6] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt, "The Geneva Convention on the Treatment of Object Aliasing," *SIGPLAN OOPS Mess.*, vol. 3, no. 2, pp. 11–16, Apr. 1992.

[7] P. Müller and A. Poetzsch-Heffter, "Modular Specification and Verification Techniques for Object-oriented Software Components," in *Foundations of Component-based Systems*, G. T. Leavens and M. Sitaraman, Eds. New York, NY, USA: Cambridge University Press, 2000, pp. 137–159.

[8] B. W. Weide and W. D. Heym, "Specification and Verification with References," in *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems*, 2001, pp. 50–59.

[9] P. W. O'Hearn, "Resources, Concurrency, and Local Reasoning," *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 271–307, Apr. 2007.

[10] C. Gordon, "Verifying concurrent programs by controlling alias interference," PhD Thesis, University of Washington, 2014.

[11] I. Filipović, P. O'Hearn, N. Torp-Smith, and H. Yang, "Blaming the client: on data refinement in the presence of pointers," *Formal Aspects of Computing*, vol. 22, no. 5, pp. 547–583, 2010.

[12] G. Kulczycki, "Direct Reasoning," PhD Dissertation, Clemson University, 2004.

[13] Y. Sun, "Towards Automated Verification of Object-Based Software with Reference Behavior," PhD Dissertation, Clemson University, 2018.

[14] N. M. J. Mbwambo, "A Well-Designed, Tree-Based, Generic Map Component to Challenge the Process Towards Automated Verification," Master's Thesis, Clemson University, 2017.

[15] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=645683.664578

[16] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky, "Effective Interactive Proofs for Higher-order Imperative Programs," in *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '09. New York, NY, USA: ACM, 2009, pp. 79–90.

[17] B. Jacobs, J. Smans, and F. Piessens, "Verifying the composite pattern using separation logic," in *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems*, ser. SAVCBS'08, 2008, pp. 83–88.

[18] F. Bobot and J.-C. Filliâtre, "Separation Predicates: A Taste of Separation Logic in First-Order Logic," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, T. Aoki and K. Taguchi, Eds. Springer Berlin Heidelberg, 2012, vol. 7635, pp. 167–181.

[19] M. Botinčan, M. Parkinson, and W. Schulte, "Separation Logic Verification of C Programs with an SMT Solver," *Electron. Notes Theor. Comput. Sci.*, vol. 254, pp. 5–23, Oct. 2009.

[20] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard, "Using First-Order Theorem Provers in the Jahob Data Structure Verification System," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Cook and A. Podelski, Eds. Springer Berlin / Heidelberg, 2007, vol. 4349, pp. 74–88, 10.1007/978-3-540-69738-1_5.

[21] R. Piskac, T. Wies, and D. Zufferey, "Automating Separation Logic with Trees and Data," in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 711–728.

[22] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local Reasoning About Programs That Alter Data Structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL '01. London, UK: Springer-Verlag, 2001, pp. 1–19.

[23] P. W. O'Hearn, H. Yang, and J. C. Reynolds, "Separation and Information Hiding," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04. New York, NY, USA: ACM, 2004, pp. 268–280.

[24] I. T. Kassios, "The Dynamic Frames Theory," *Form. Asp. Comput.*, vol. 23, no. 3, pp. 267–288, May 2011.

[25] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Regional Logic for Local Reasoning About Global Invariants," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 387–411.

[26] A. Banerjee, M. Barnett, and D. A. Naumann, "Boogie Meets Regions: A Verification Experience Report," in *Proceedings of the 2Nd International Conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 177–191.

[27] D. E. Harms and B. W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components," *IEEE Trans. Softw. Eng.*, vol. 17, no. 5, pp. 424–435, May 1991.

[28] C. T. Cook, "A Web-Integrated Environment For Component-Based Software Reasoning," Master's Thesis, Clemson University, 2011.

[29] A. Weide, P. A. G. Sivilotti, and M. Sitaraman, "Array abstractions to simplify reasoning about concurrent client code," Clemson University - School of Computing, Clemson, SC 29634, Tech. Rep. RSRG-17-05, November 2017.

[30] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 97–116.

[31] A. Weide, P. A. G. Sivilotti, and M. Sitaraman, "Enabling Modular Verification with Abstract Interference Specifications for a Concurrent Queue," in *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Blazy and M. Chechik, Eds., vol. 9971, 2016, pp. 119–128.

[32] S. T. Taft, "Multicore programming in ParaSail," in *International Conference on Reliable Software Technologies*. Springer, 2011, pp. 196–200.

[33] G. Kulczycki, H. Smith, H. Harton, M. Sitaraman, W. F. Ogden, and J. E. Hollingsworth, "The Location Linking Concept: A Basis for Verification of Code Using Pointers," in *Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, R. Joshi, P. Müller, and A. Podelski, Eds. Springer Berlin Heidelberg, 2012, vol. 7152, pp. 34–49.