Towards a Modular Proof Rule for Parallel Operation Calls*

Alan Weide[1], Paolo A. G. Sivilotti[1], and Murali Sitaraman[2]

[1] The Ohio State University, Columbus OH 43221, USA,
weide.3@osu.edu, paolo@cse.ohio-state.edu
[2] Clemson University, Clemson SC 29634, USA,
murali@clemson.edu

Abstract. The specific focus of this paper is on specification-based proof rules for parallel operation calls to facilitate modular verification. The rules proposed are novel in that they raise and address the following complications, without compromising soundness or relative completeness: situations where operation specifications are relational, situations when preconditions for only some serializations might hold, and situations where a software developer explicitly lists the serializations to be considered to minimize what needs to be verified. The paper also contains a sketch of a soundness proof, and illustrates the ideas and rules with examples.

1 Introduction

As computing hardware becomes ever more parallel, the importance of making sure software written for it is correct—and the difficulty thereof—becomes increasingly obvious. It is well-known to software developers that writing, testing, debugging, and maintaining parallel software is exponentially harder than doing so for sequential software of similar scale. The interaction of various threads with one another cause problems to manifest unpredictably, frustrating traditional testing and debugging techniques. Therefore, for high confidence systems, ideally software should be verified to be correct with respect to its specification. Moreover, verification needs to be done in a modular fashion, one component at a time, in order to be scalable. For modular verification, parallel programs should be composed from components encapsulating data abstractions.

There are many active projects that are working towards a solution to the larger parallel program verification problem. Broadly speaking, there are three main areas of formal methods research that are closely related to this work.

1. Verification of full functional correctness of programs that make use of rich data abstractions. Such research is usually applied to sequential programs because of complications to concurrency introduced by abstraction.

* Tech Report OSU-CISRC-3/20-TR01

2. Verification of full functional correctness of concurrent programs. Most of this research is dedicated to verifying the correctness of programs that make use of concurrency primitives or other low-level software components.
3. Verification of a limited set of properties (such as safety, termination, or data race freedom) of concurrent and parallel programs. Frequently, this research deals with programs that do not make use of data abstractions but occasionally encompasses higher-level programs. Efforts in this space are employed either at compile-time or run-time.

Substantial work appears in any one or at the intersection of any two of these research areas. For example, the intersection of (1) and (3) includes work that uses data abstraction in concurrency, but only verifies certain properties of those programs. The focus of our larger project is at the intersection of all three areas, that is, the automated and formal verification of full functional correctness of parallel programs with rich data abstractions.

To achieve the larger objective of modular verification, the present paper focuses on a step-by-step approach for development of a proof rule for operation calls. One complication arises when the precondition for only one of the many possible serializations may be true, and another when specifications of some of the operations reused in the program are relational. Such specification of *relational behavior*, when multiple outputs are specified as possible for a given set of inputs, arises routinely, for example, in the description of optimization problems. The complication here is that when these are parallel programs, not only must the formal reasoning deal with the possibility of nondeterminism being introduced by the parallel execution itself, but also by the specificational indeterminacy of the operations that are called. Yet another complication is perhaps the most obvious in that just too many serializations will have to be considered in the verification process. The proposed rules handle operations calls with nontrivial preconditions and relational behavior, and allow a software developer to dictate if a small subset of serializations (instead of every one) can be considered in verification.

2 Related Work

2.1 Verification of Correctness of Sequential Programs

One central challenge in sequential, object-oriented software verification involves objects, aliasing, and properties about the heap [15, 28]. Separation logic is an extension of Hoare's logical rules to address these challenges. Examples of verification using separation logic in Coq include [6, 5] and in VeriFast to verify Java and C programs include [11, 12]. Automating verification with separation logic is the topic of [2, 4, 24, 25] and others.

2.2 Verification of Correctness of Concurrent Programs

Some of the foundational ideas in the verification of concurrent and parallel programs were introduced by Owicki and Gries in their seminal paper [23]. It

provides foundations for using Hoare logic to verify the partial correctness of parallel programs in a simple parllel programming language. As with most research in this domain, Owicki and Gries prove the correctness of parallel programs at an extremely low level (e.g., at the level of locks and semaphores, with variables being simple integers), without regard for data abstraction.

In addition to verifying the correctness of sequential programs, separation logic has been used to some effect for concurrent programs. Concurrent separation logics have expanded the capabilities of (sequential) separation logic by adding rules for reasoning about concurrent programs [22]. An important new direction for abstraction in concurrent separation logic is the focus of [14].

In [17], Leino discusses the joining of specification statements, and provides a brief discussion of the joining of specification statements in the face of parallelism. Specifically, the join operator $\sqcup$ in [16] is the least-refined statement that refines both (all) of the parallel statements, and is similar in spirit to the construction of the proof rules here.

2.3 Verification of Certain Properties of High-Level Programs

Much of the work in the verification of concurrent programs is focused not full functional correctness, but rather on specific properties (*e.g.*, memory safety, termination, data race freedom) hold.

In Deterministic Parallel Java [3] (DPJ), regions are defined explicitly by the programmer. These annotations allow a DPJ compiler to guarantee, statically, that two concurrent operations are data race free and thus will produce a deterministic result. ParaSail [33] is a new programming language that relies on value semantics to verify that concurrent statements are non-interfering.

Both the above approaches are limited by what can be syntactically checked, and ParaSail in particular is limited by the fact that objects are not subdivided into regions that can be reasoned about independently.

The determinism guarantees in this work and others amount to showing *data race freedom*, though the present approach deals with high-level programming constructs. There is a large body of work on showing low-level race freedom, including both static approaches [7, 20, 1, 10, 9, 19, 13] and dynamic ones [27, 26, 8, 29]. Like DPJ and ParaSail, these are limited to guaranteeing some level of determinism and do not claim to formally verify full functional correctness.

3 Context and Background

The context for the present work is RESOLVE [31, 30], an integrated programming and specification language that enables verification of full behavioral correctness of sequential software components to encapsulate rich data abstractions [31]. RESOLVE specifications use an extensible collection of mathematical models and they allow alternative implementations. Verification of code that uses components proceeds in a modular fashion, using only the specifications of those

components. Given code annotations, such as invariants and pre- and postconditions, code verification is mostly automated, and there have been efforts to build easy-to-use tools to use it for verified software [30, 36]. The present effort builds on this prior work in sequential software verification.

In particular, this work is focused on *fork-join parallel* programs, wherein several threads are spawned at some point, each to perform a unit of work (called the *parallel section*), and the main thread does not continue until all of its children have finished their work (i.e., at the end of the parallel section). This execution model does simplify some aspects of this work, but it does not restrict its utility—the results from this paper could be generalized to a number of different distributed and parallel computing execution models.

3.1 Abstractions to Simplify Reasoning

In RESOLVE (as in other sequential verification languages), the specification for a data abstraction (called the *concept*) is decoupled from its implementations. A concept contains a *math model*, which is a purely mathematical interpretation of the data type. An implementation (called a *realization*) provides a mapping from its concrete state to the abstract state based on that math model. In turn, the concrete state of a realization is based on the *abstract* values of its fields.

We consider a specification of trees. One way to capture a linked tree structure in the heap in separation logic is by the use of predicates, such as the one below [21]:

$$\begin{aligned} \mathit{tree}(E, \tau) \iff & \textbf{if } (\text{isAtom?(E)} \land E = \tau) \textbf{ then } \mathit{emp} \\ & \textbf{else } \exists xy\tau_1\tau_2.\tau = \langle \tau_1, \tau_2 \rangle \land (E \mapsto [l : x, r : y] * \mathit{tree}(x) * \mathit{tree}(y)), \end{aligned}$$

Predicate definitions as abstraction in separation logic, while powerful and necessary in verifying an underlying tree implementation, ideally would be shielded through a data abstraction specification to enable reasoning about code based solely on the *abstract* value of an object, rather than its implementation. In [18], for example, a "shared" Tree is modeled mathematically without implementation details and operations allow navigation of a tree and manipulation of its contents. The specification makes no reference to underlying linked structures used in implementing the trees, so it is possible to view trees in mathematical terms for the purposes of reasoning of client code. One-time verification of tree implementations do require non-trivial reasoning [32] as in the case of using separation logic, but that reasoning is compartmentalized within implementations.

3.2 Specifications of Operations with Functional and Relational Behavior

There are two classes of operations we consider when writing these proof rules: operations with *functional* specifications, and those with *relational* specifications. Functional operations have exactly one possible output for any given input (e.g., the operation Dequeue() in the listing below). Relational operations, on

the other hand, might have several possible outputs for a given input (e.g., the operation InsertAnywhere() below). Here, a Queue is conceptualized as a mathematical string (similar to a sequence), so the pre- and postconditions are written in terms of operations on mathematical strings: $|q|$ is the length of string q, $\langle x \rangle$ is the singleton string containing the entry x, $\circ$ is the concatenation operator, and $q[a, b)$ is the substring of q beginning at index a and ending the entry before index b.

operation Dequeue(**updates** q: Queue, **replaces** x: Item)
requires $|q| > 0$;
ensures $\#q = \langle x \rangle \circ q$;

operation InsertAnywhere(**updates** q: Queue, **clears** x: Item)
ensures $\exists(i)(q = \#q[0, i) \circ \langle x \rangle \circ \#q[i, |\#q|))$;

If all of the operations in the parallel section of a program have functional specifications and there are no data races, then the order of their execution cannot matter—any serialization of those operations *must* result in the same output, and that output is the result of any given sequential execution of those operations. In the case of relational specifications, the semantics are less clear. For example, consider what happens when the two operations in the listing above are run in parallel (assume they are data race free). If we take the serialization Dequeue(q, x); InsertAnywhere(q, y), we can conclude, at a minimum, that the element removed by Dequeue is not the element added by InsertAnywhere. However, if we take the other serialization (that is, InsertAnywhere(q, y); Dequeue(q, x)), we have that the element removed by Dequeue might in fact be the element added by InsertAnywhere. Proof rules related to these lines of reasoning are found in Sections 4.1 and 4.2.

3.3 Reasoning About Non-Interference Between Threads

When a parallel program executes, there is the possibility that several threads might try to write to the same memory location during the parallel section (or, one thread might read a memory location that another writes). When such a situation (known as a *data race*) occurs, we say the threads are *interfering*, and the execution of the program is not "safe" because a data race can introduce nondeterminism to the output. Most fork-join parallel algorithms rely on having a deterministic output (e.g., array summation, searching a tree), so it is desirable to enforce determinism statically, at verification-time.

Our prior work on showing the non-interference of parallel operations relies on the use of a novel notion of *non-interference contracts* [35]. A non-interference contract divides the abstract state of an object into *partitions*, which are disjoint. The non-interference contract then provides *effects clauses* for each operation; it provides a *partition mode* for each partition of this non-interference contract (one of **affects**, **preserves**, or **ignores**).

A client of a non-interference contract uses effects clauses to establish that several parallel calls to operations which share parameters are *non-interfering* by

showing that any partition which is in affects-mode in one operation is ignored by all others in the parallel block. Importantly (for preserving abstraction), partition modes may be conditional on the *abstract value* of a parameter.

On the implementation side, it must be shown that every method body actually respects these partition modes. To facilitate this, the programmer provides an *interference correspondence* which places each *concrete* field (or, more specifically, each *partition* of each concrete field) of the component into one of the (abstract) partitions provided by the non-interference contract this implementation is said to respect. Membership of a concrete field in a partition may be conditional on the abstract values of the concrete fields of the component.

A deeper introduction to non-interference contracts and how they ensure data race freedom can be found in [34, 35].

4 Modular Proof Rules for Safe Parallel Execution

The current phase of this work deals only with *non-interfering* processes—that is, processes that are data race free. In the programs we examine, non-interference comes in several flavors. First, two processes are non-interfering if they share no variables in common. This is surprising, of course, because it fails to account for aliasing of references. However, because RESOLVE has clean semantics (each variable is the unique owner of whatever memory it refers to), the need to deal with aliases is eliminated in all but the most complex of cases, which are dealt with separately [32]. Second, two processes are non-interfering if they do share a variable, but it is only read from—not written to—by both processes. A read-only variable is denoted by the **preserves** parameter mode in RESOLVE. Finally, two processes can be non-interfering even when they share and modify a variable through the use of *non-interference contracts* [35]. Two threads are non-interfering (according to a non-interference contract) if every partition that is affected by one thread is ignored by the other.

When two parallel threads do not interfere with one another (i.e., they have no data races), their execution is equivalent to a serialization of their constituent instructions. In fact we can say something stronger, that their execution is equivalent to *all possible serializations* of their constituent instructions. That is, when reasoning about the behavior of a parallel section of a program, its behavior is entirely deterministic (absent explicit nondeterminism in the specification or implementation). This insight allows us to simplify proof rules for parallel programs to make it easy and fast to verify their correctness.

We use the notation $S_1 \ddagger S_2$ to mean "S_1 is non-interfering with S_2".

4.1 Operations with Functional Specifications

When the specifications of operations inside the parallel section of a program are functional, the semantics of their concurrent composition is somewhat trivial—at least when the operations are non-interfering. For an informal proof of this statement, consider that if there are no data races (that is, there is no part of

memory that is written to by more than one thread, nor is read by a thread if it is written to by another) between the two overall operations, then any two constituent instructions comprising the operations could be executed in either order and still produce the same program state. Therefore, no matter which interleaving of these instructions we choose (the actual execution of these operations might "choose" any of them), the resulting state of the program will be the same. So we have that the *concurrent* composition of some data race free operations is exactly equivalent to any of the *sequential* compositions of the same operations (all of which are equivalent to each other). Therefore, the proof rule for the concurrent composition of (two) non-interfering operations with functional behavior is identical to the proof rule for the sequential composition of those operations (here named P and Q).

operation P(**preserves** x: T1, **updates** y: T2)
  ensures $y = F(x, \#y)$;

operation Q(**preserves** x: T1, **updates** y: T2)
  ensures $y = G(x, \#y)$;

cobegin
  P(u, v);
  Q(u, v);
end;

$$\frac{\{Pre\}P(u,v);Q(u,v)\{Post_{PQ}\} \quad P(u,v) \ddagger Q(u,v)}{\{Pre\}P(u,v) \| Q(u,v)\{Post_{PQ}\}} \tag{1}$$

In our notation for proof rules, we define $Post_{PQ}$ to be the postcondition of the sequential piece of code $P(u,v);Q(u,v)$. When P and Q are as specified above, $Post_{PQ} \equiv y = G(x, F(x, y_0))$.

When operations with functional specifications are composed in parallel, because all possible interleavings (including both serializations of the operations) result in the same state of the program, it is sufficient to consider just one of those interleavings and use it to verify our program. Specifically, we consider the serialization of the two operations in the order they are listed in the program.

4.2 Operations with Relational Specifications

Operations with relational specifications complicate verification considerably, but not insurmountably. Consider the argument provided above in Section 4.1 that no matter which interleaving of the operations is chosen at runtime, the output will be identical because they are data race free. We can use this information to conclude something *stronger* about the concurrent composition of several non-interfering relational operations than we could about any of their sequential compositions. Specifically, the proof rule for the parallel composition of (two) non-interfering operations with relational specifications (here named P and Q) is as follows, provided the calls to P(u, v) and Q(u, v) are non-interfering.

operation P(**preserves** x: T1, **updates** y: T2)
  ensures $Post_P(x, \#y, y)$;

operation Q(**preserves** x: T1, **updates** y: T2)
  ensures $Post_Q(x, \#y, y)$;

cobegin
  P(u, v);
  Q(u, v);
end;

$$\frac{\begin{array}{l}\{Pre\}P(u,v);Q(u,v)\{Post_{PQ}\}\\ \wedge\{Pre\}Q(u,v);P(u,v)\{Post_{QP}\}\end{array} \quad P(u,v) \ddagger Q(u,v)}{\{Pre\}P(u,v) \| Q(u,v)\{Post_{PQ} \wedge Post_{QP}\}} \tag{2}$$

That is, consider *both* (or, in the general case, *all*) possible sequential orderings (in this case $P(u,v);Q(u,v)$ and $Q(u,v);P(u,v)$) of the operations. The output of their concurrent execution is a state that is reachable by both (all) of them.

Perhaps unsurprisingly, when P and Q are both functional, an application of the relational proof rule is equivalent to an application of the functional proof rule because, as shown below in Section 4.4, non-interfering statements necessarily commute (so $Post_{PQ} \equiv Post_{QP}$).

4.3 Operations with Differing Preconditions

Conspicuously absent from the proof rules (1) and (2) are a discussion of preconditions for the operations in a parallel section. In part, this is because the properties to be established before a parallel section are perhaps counterintuitive.

To formulate the proof rule for the parallel composition of statements with differing preconditions, first we augment the specifications of P and Q:

operation P(**preserves** x: T1, **updates** y: T2)
  requires $Pre_P(x, y)$;
  ensures $Post_P(x, \#y, y)$;

operation Q(**preserves** x: T1, **updates** y: T2)
  requires $Pre_Q(x, y)$;
  ensures $Post_Q(x, \#y, y)$;

The following proof rule describes the semantics of the parallel execution of two non-interfering statements with differing preconditions:

$$\frac{\begin{array}{l}\{Pre_{PQ}\}P(u,v);Q(u,v)\{Post_{PQ}\}\\ \wedge\{Pre_{QP}\}Q(u,v);P(u,v)\{Post_{QP}\}\end{array} \quad P(u,v) \ddagger Q(u,v)}{\{Pre_{PQ} \vee Pre_{QP}\}P(u,v) \| Q(u,v)\{Post_{PQ} \wedge Post_{QP}\}} \tag{3}$$

Similar to $Post_{PQ}$, Pre_{PQ} is the weakest precondition (according to the specification) that must be satisfied before verification of the sequential code $P(u, v); Q(u, v)$ can proceed.

That is, *either* (any) of the preconditions for the serializations must be met in order to conclude *both* (all) of the postconditions. This can result in surprising behavior. Consider the following two non-interfering operations for Natural, the programming type that is modeled by the natural (i.e., non-negative) numbers:

operation Increment(**updates** n: Natural)
  ensures $n = \#n + 1$;

operation Decrement(**updates** n: Natural)
  requires $n > 0$;
  ensures $\#n = n + 1$;

The parallel composition of these two statements operating on the same variable might look like the following:

cobegin
  Increment(n);
  Decrement(n);
end;

Applying various rules for sequential composition of statements, we have the following two Hoare triples representing the semantics of each serialization:

$$\{true\}\text{Increment(n); Decrement(n)}\{n = n_0\}$$
$$\{n_0 > 0\}\text{Decrement(n); Increment(n)}\{n = n_0\}$$

When Increment(n) and Decrement(n) are non-interfering[3], the semantics of their parallel execution based on (3) is as follows:

$$\{true\}\text{Increment(n)}\|\text{Decrement(n)}\{n = n_0\}$$

The execution of one of the serializations has a nontrivial precondition $n > 0$ while the other one has the trivial precondition *true*; by proof rule (3), the precondition for their parallel composition therefore is *true*. But the scheduler might ("demonically") choose to execute these two operations in exactly the order Decrement(n); Increment(n), and a violated precondition would result. However, because of the non-interference of these two statements, the resultant state of the the program must be identical whether the scheduler chose that serialization, the other one (with the precondition *true*), or even an arbitrary interleaving of the instructions comprising the two operations. Therefore, it can be assumed that the scheduler made an "angelic" choice—that is, the serialization Increment(n); Decrement(n).

[3] An implementation of the type Natural for which the operations Increment(n) and Decrement(n) are non-interfering is left as an exercise to the reader.

Example Application of Proof Rule. As an example, consider the following operation specifications and **cobegin** statement (which is correct according to the proof rule). For now, we assume the statements are non-interfering.

operation AddAnEnd(**updates** q: Queue, **clears** x: Item)
  ensures $q = \#q \circ \langle \#x \rangle \vee q = \langle \#x \rangle \circ \#q$;

operation RemoveAnEnd(**updates** q: Queue, **replaces** x: Item)
  requires $|q| > 0$;
  ensures $\#q = q \circ \langle x \rangle \vee \#q = \langle x \rangle \circ q$;

assume $q = \langle 10, 20, 30 \rangle \wedge u = 4$;
cobegin
  AddAnEnd(q, u);
  RemoveAnEnd(q, v);
end;
confirm
  $(q = \langle 4, 10, 20 \rangle \wedge v = 30) \vee$
  $(q = \langle 20, 30, 4 \rangle \wedge v = 10)$;

First, the precondition for either serialization is met by these initial conditions, so we can proceed. The reachable states for the two possible serializations of these operations are in Table 1.

Serialization	Reachable States
AddAnEnd(q, u); RemoveAnEnd(q, v);	$q = \langle 4, 10, 20 \rangle \wedge v = 30$
	$q = \langle 20, 30, 4 \rangle \wedge v = 10$
	$q = \langle 10, 20, 30 \rangle \wedge v = 4$
RemoveAnEnd(q, v); AddAnEnd(q, u);	$q = \langle 4, 10, 20 \rangle \wedge v = 30$
	$q = \langle 20, 30, 4 \rangle \wedge v = 10$
	$q = \langle 4, 20, 30 \rangle \wedge v = 10$
	$q = \langle 10, 20, 4 \rangle \wedge v = 30$

Table 1. Reachable states for the two orderings of AddAnEnd(q, u) and RemoveAnEnd(q, v), with matching ones highlighted.

The intersection of these two sets of states are exactly the **confirm** statement in the listing above:

$$q = \langle 20, 30, 4 \rangle \wedge v = 10$$
$$q = \langle 4, 10, 20 \rangle \wedge u = 0 \wedge v = 30$$

4.4 Sketch of Proof of Soundness

The full proofs of soundness of these proof rules are lengthy (due to the scaffolding needed to talk about high-level semantics), so we give a sketch of the proof here. The are a few key concepts behind this proof. The first is *clean semantics*, a property of a programming language which prohibits aliasing and ensures that any two distinct variables in scope refer to distinct portions of memory. The second is that a given state of memory (or concrete state of a variable) abstracts to a single value.[4]

We define an *operation* as a sequence of actions, each of which "targets" a set of partitions (here, a partition is simply a unique set of memory locations). The partitions are either *preserved* (read from) or *affected* (written to) by an action. The definition of non-interference can be framed in these terms: two operation calls are non-interfering if there is no partition that is an affected target of some action of one operation that is either an affected or preserved target of any action in the other operation.

Given two actions a_{O1} and b_{O2} of non-interfering operation calls $O1$ and $O2$, we know that there is no memory location that is written to by one action that is read from or written to by the other (because otherwise these operations calls wouldn't be non-interfering). Therefore, if memory starts in some state S, and the state of memory after executing actions a_{O1} followed by b_{O2} is S', and the state of memory after executing actions b_{O2} followed by a_{O1} is S'', then $S' = S''$. Thus, the state of memory does not depend on what order these actions are executed in, so we can assume either one.

This can be generalized to encompass the entirety of the operations $O1$ and $O2$: the state of memory will be identical no matter what interleaving of the operations' constituent actions actually happens at run time. Because a given state of memory abstracts to a particular value (or set of values, if an abstraction *relation* is involved), the set of possible abstractions of memory, then, is the same no matter what interleaving actually happens at run time. Therefore, because the state of memory after the concurrent execution of two non-interfering actions *must* be a state that is reachable by all possible interleavings (including either serialization), its abstraction must also be reachable by all possible interleavings.

Therefore, if there are two calls to operations with relational specifications that are non-interfering, it must be the case that the state of the program after their concurrent execution is one that is reachable by both serializations; hence, the relational proof rule (2) is sound. The functional proof rule (1) is a special case of the relational proof rule, so it is also sound. The justification for the weakened precondition is given alongside rule (3).

[4] RESOLVE also has provisions for *abstraction relations* where a single concrete state maps to some set of abstract values rather than a single value. Whenever in the course of the proof this possibility needs accounting for, it is mentioned.

4.5 Conditional Non-Interference

The effects in a non-interference contract might be conditional on the abstract values of parameters, in which case the actual non-interference of several statements might also be conditional on those values. The proof rule for such a situation is a complication of the proof rules presented so far. Consider the operations P and Q, augmented with viable effects clauses (suppose a variable of type T has two partitions, a and b).

operation P(**updates** x: T)
  affects $x@a$;
  when $G_P(x)$ **affects** $x@b$;
  requires $Pre_P(x)$;
  ensures $Post_P(\#x, x)$;

operation Q(**updates** x: T)
  affects $x@b$;
  when $G_Q(x)$ **affects** $x@a$;
  requires $Pre_Q(x)$;
  ensures $Post_Q(\#x, x)$;

Parallel calls to these operations are non-interfering only when $\neg G_P(x)$ (otherwise, both statements affect $x@b$) or $\neg G_Q(x)$ (otherwise, both statements affect $x@a$). A proof rule with support for conditional effects must include these guards in its assessment of a valid precondition for the parallel composition of the statements.

$$\frac{\begin{array}{c}\{Pre_{PQ}\}P(u);Q(u)\{Post_{PQ}\} \\ \wedge\{Pre_{QP}\}Q(u);P(u)\{Post_{QP}\}\end{array} \quad G_{PQ} \Rightarrow P(u) \ddagger Q(u)}{\{G_{PQ} \wedge (Pre_{PQ} \vee Pre_{QP})\}P(u) \| Q(u)\{Post_{PQ} \wedge Post_{QP}\}} \tag{4}$$

When the effects of operations are conditional, the non-interference of two statements is determined by the abstract values of the arguments *before* the parallel section.

Soundness of Conditional Non-Interference. This proof rule is sound given the non-conditional version is sound. If G_{PQ} is satisfied, then $P(u) \ddagger Q(u)$, and the rule reduces to the non-conditional version.

5 Specialization: Considering Subsets of Serializations

5.1 A Single Serialization

It should be noted that while rule (2) is the *strongest* proof rule for operations with relational specifications that can be derived from the guarantees made by non-interference, it is not the only useful one. For example, an alternative to the

relational proof rule is one where only a single serialization is considered (P and Q are as defined in Section 4.2):

$$\frac{\{Pre\}P(u,v);Q(u,v)\{C\} \quad P(u,v) \ddagger Q(u,v)}{\{Pre\}P(u,v)\|Q(u,v)\{C\}} \tag{5}$$

The weaker rule is useful because of a surprising fact: employing the strong rule (2) breaks the relative completeness of our proof system. Consider the example above with the two operations AddAnEnd and RemoveAnEnd, and the implementation of the operation DoubleRotate below. Conceptually, DoubleRotate "shifts" the queue two entries either forward or backward.

```
operation DoubleRotate(updates q: Queue, updates x: Item);
    requires 1 < |q|;
    ensures
        (⟨x⟩ = (#q ∘ ⟨#x⟩)[1, 2) ∧ q = (#q ∘ ⟨#x⟩)[2, |#q|) ∘ #q[0, 1))∨
        (⟨x⟩ = (⟨#x⟩ ∘ #q)[|#q| − 1, |#q|)∧
            q = #q[|#q| − 1, |#q|) ∘ (⟨#x⟩ ∘ #q)[0, |#q|));
procedure
    var y: Item;
    cobegin
        AddAnEnd(q, x);
        RemoveAnEnd(q, y);
    end;
    AddAnEnd(q, y);
    RemoveAnEnd(q, x);
end DoubleRotate;
```

If AddAnEnd and RemoveAnEnd are non-interfering, this is a correct implementation of the operation as specified. It is correct because any non-interfering implementations of those operations would necessarily operate on opposite ends of the queue. Further, the implementations used couldn't possibly change over the course of the body, so the two calls to AddAnEnd on lines 11 and 14 must necessarily operate on the same end of the queue. However, with the proof rule for the **cobegin** statement presented in Section 4.2 in conjunction with the proof rule for sequential exeuction (for lines 14 and 15), the verifier could not prove this procedure body correct. The difficulty here is the fact that the verifier does not (and cannot, if the locality of reasoning is to be preserved), use in its verification of the sequential portion of the code the additional information afforded to it by virtue of the non-interference of the two operation calls.

Clearly, the procedure body above would not verify with rule (5), either. However, under the semantics presented in this section (that is, the concurrent execution of two non-interfereing operations is identical to their sequential composition in the order in which they are written), the procedure body is not, in fact, correct (and cannot be proved so), while under the semantics based on (2) it *is* a correct body, and still it cannot be proven correct. Hence, the stronger proof rule does not preserve relative completeness while the simpler one does.

Rule (5) is identical to the proof rule introduced above for operations with functional behavior. This is an additional practical benefit to employing the weaker rule over the strong one, (2). Since the two proof rules are identical, the need to determine whether a particular operation has functional or relational specifications (itself an undecidable question) is eliminated. Ultimately, determination of a proper rule may require additional discussion, analysis, and possible experimentation.

5.2 Consideration of Subsets of Serializations

Efficiently automating reasoning about parallel code poses a core challenge: as the number of operations inside the **cobegin** block grows, the number of possible serializations to consider in the relational case grows extremely fast. To automate this reasoning process, then, clearly we need a shortcut or heuristic to approximate the results from this exponential explosion of possibilities. The proposed solution is to allow the programmer to limit the number of interleavings that should be considered for automated verification through an annotation. For example, consider the following parallel block of code:

```
cobegin
  P(x);
  Q(y);
  R(z);
  S(w);
end;
```

There are 24 possible sequential orderings of the statements in the code above. [5] That means a verifier would, in effect, have to verify the same piece of code 24 times! This is clearly not scalable, so we introduce a **considering** annotation which is followed by a set of sequential orderings that should be considered to extract enough information from the composition of these operations to prove, e.g., a procedure's postcondition. Suppose that the programmer can determine (through manual reasoning of their own code) that considering only the written ordering and its reverse will allow the verifier to conclude what it needs. Then, the programmer-annotated cobegin block would look like the following.

```
cobegin
considering <1, 2, 3, 4>, <4, 3, 2, 1>
  1: P(x);
  2: Q(y);
  3: R(z);
  4: S(w);
```

[5] Of course, even considering just the serializations of the operations is an approximation, because it does *not* take into account "interleavings" of the internal code for these operations wherein the first instruction of P is executed, then the first instruction of Q, and so on. Accounting for interleavings not only would cause further explosion but would also violate the principles of abstraction and modularity.

end;

The **considering** annotation here tells the verifier, "use only the information you can conclude from the serializations P(x); Q(y); R(z); S(w); and S(w); R(z); Q(y); P(x); in proving the correctness of this program." Such an annotation can prevent an exponential buildup of possible serializations to consider while still affording the verifier additional information from the non-interference of threads when it is applicable.

Soundness of the Considering Annotation. Because the "full" proof rule (2) above requires showing the *conjunction* of the reachable states of all possible serializations of a **cobegin** statement's constituent operations, and since that is sound, any rule derived from the **considering** annotation is also sound, since the conjunction of several clauses implies each of those clauses (and any sub-conjunction of them).

6 Conclusions and Future Directions

Verifying the correctness of high-level concurrent programs is a challenging problem, made more complex by relational specifications for operations. The proof rules presented here provide a sound, relatively complete step toward the verification of fork-join parallel programs which make use of data abstraction.

The proof rules presented here are *client-side* proof rules; that is, the implementation is disregarded and assumed to be correct. However, the implementation must also be proven correct with respect to its specification (in this case, its *non-interference contract*). In RESOLVE, a realization destined for concurrency provides a *non-interference correspondence* that maps the concrete state to the partitions defined in its interference contract. In particular, the non-interference correspondence maps *partitions* of concrete fields to partitions of the interference contract, and the verification that a particular procedure body respects the effects in the interference contract proceeds based only on (a) the behavioral and non-interference specifications of the constituent statements and (b) the interference correspondence. If the effects clause is met by all procedure bodies in the realization, then the non-interference contract can be used with confidence.

The technique presented here can be extended and generalized to encompass parallel iteration and parallel programming styles beyond fork-join. However, work is ongoing to explore how to achieve tractable verification in these situations while maintaining the abstraction and modularity properties discussed herein.

Specifically, it is likely that non-interference contracts will need to be augmented to handle more complicated situations (e.g., *atomic*, *owns*, or *locks/unlocks* annotations to deal with various concurrency models, and *segmented partitions* to handle parallel loops and more complex data abstractions).

The rules presented here will be used to implement a **cobegin** statement in the RESOLVE programming language.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, Mar. 2006.
2. F. Bobot and J.-C. Filliâtre. Separation predicates: A taste of separation logic in first-order logic. In T. Aoki and K. Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin Heidelberg, 2012.
3. 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*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.
4. M. Botinčan, M. Parkinson, and W. Schulte. Separation logic verification of c programs with an smt solver. *Electron. Notes Theor. Comput. Sci.*, 254:5–23, Oct. 2009.
5. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.
6. 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*, ICFP '09, pages 79–90, New York, NY, USA, 2009. ACM.
7. D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.
8. C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.
9. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM.
10. S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.
11. 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*, SAVCBS'08, pages 83–88, 2008.
12. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.
13. M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala. Deterministic parallelism via liquid effects. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 45–54, New York, NY, USA, 2012. ACM.
14. R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In H. Yang, editor, *Programming Languages and Systems*, pages 696–723, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

15. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, June 2007.
16. K. M. Leino and R. Manohar. Joining specification statements. *Theoretical Computer Science*, 216(1):375 – 394, 1999.
17. K. R. M. Leino. Specification and verification of object-oriented software. In M. Broy, W. Sitou, and T. Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, Volume 22 NATO Science for Peace and Security Series - D: Information and Communication Security, pages 231–266. IOS Press, 2009.
18. N. M. J. Mbwambo. A well-designed, tree-based, generic map component to challenge the progress towards automated verification. Master's thesis, Clemson University - School of Computing, May 2017. 122 pages.
19. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 327–338, New York, NY, USA, 2007. ACM.
20. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.
21. P. O'Hearn. A primer on separation logic (and automatic program verification and analysis). 05 2012.
22. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, Apr. 2007.
23. S. Owicki and D. Gries. *An Axiomatic Proof Technique for Parallel Programs*, pages 130–152. Springer New York, New York, NY, 1978.
24. R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 773–789, Berlin, Heidelberg, 2013. Springer-Verlag.
25. 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*, pages 711–728, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
26. E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.
27. E. Pozniansky and A. Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded C++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007.
28. 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*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
29. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.
30. 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*, 23(5):607–626, 2011.
31. M. Sitariman and B. Weide. Component-based software using resolve. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, Oct. 1994.
32. Y.-S. Sun. *Towards Automatd Verification of Object-Based Software with Reference Behavior*. Phd thesis, Clemson University, School of Computing, 2018.
33. T. Taft and J. Hendrick. Designing ParaSail, a new programming language. http://parasail-programming-language.blogspot.com/, 2017.
34. A. Weide, P. A. G. Sivilotti, and M. Sitaraman. Enabling modular verication of concurrent programs with abstract interference contracts. Technical Report RSRG-16-05, Clemson University - School of Computing, December 2016.
35. A. Weide, P. A. G. Sivilotti, and M. Sitaraman. Enabling modular verification with abstract interference specifications for a concurrent queue. In S. Blazy and M. Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 119–128, Cham, 2016. Springer International Publishing.
36. D. Welch. *Scaling Up Automated Verification: A Case Study and a Formalization IDE for Building High Integrity Software*. PhD thesis, Clemson University, School of Computing, 2019.