Separating Functional and Parallel Correctness using Nondeterministic Sequential Specifications

Jacob Burnim, George Necula, and Koushik Sen
Department of Computer Science, University of California, Berkeley
{jburnim,necula,ksen}@cs.berkeley.edu

Abstract

Writing correct explicitly-parallel programs can be very challenging. While the functional correctness of a program can often be understood largely sequentially, a software engineer must simultaneously reason about the nondeterministic parallel interleavings of the program's threads of execution. This complication is similarly a challenge to automated verification efforts.

Thus, we argue that it is desirable to decompose a program's correctness into its sequential functional correctness and the correctness of its parallelization. We propose achieving this decomposition by specifying the parallel correctness of a program with a *nondeterministic* but *sequential* version of the program. In particular, if a software engineer annotates the intended *algorithmic nondeterminism* in a program, then the program can act as its own specification in verifying the correctness of its parallelism. We can interpret the annotated program as *sequential* but *nondeterministic*, and then verify the correctness of the parallelism by showing that it introduces no additional nondeterminism.

1 Introduction

Writing correct multithreaded programs is very difficult, in large part because most of a programmer's attention is focused on the functional correctness of a sequential interpretation of their code. It is very hard to consider at the same time the additional behaviors from all possible interleavings during a parallel execution. Thus, all too often the parallel correctness of a program is an afterthought. We share a widespread belief [3] that the only way to make multithreaded programming accessible to a large number of programmers is to come up with programming paradigms and associated tools that simplify reasoning about parallel correctness and enable reasoning about functional correctness in a sequential or nearly-sequential way.

Automated parallelization work such as [13] is the most direct expression of this strategy. The major difficulty is to parallelize a program to achieve good performance while remaining completely faithful to its sequential semantics. An alternate approach is to verify that an existing parallel program implements a sequential or nearly-sequential specification. Examples of recent work along this line are atomicity analyses such as [14, 9, 10, 17, 8, 18, 11, 7]. However, both automated parallelization and analysis of parallel programs become ineffective when the synthesis or the verification of the parallelization of the code becomes entangled with the functional correctness of the program.

We recognize that parallelization correctness depends on and extends sequential functional correctness, which is a very hard problem on its own. We believe that in order to break this gridlock and make progress, it is useful to decompose correctness into functional correctness and parallelization correctness.

Our Approach. We aim to separate the correctness of the parallelization of a program from its functional correctness, with the ultimate goal of providing effective tools for verification of the parallelization, while allowing the programmer to concentrate on the functional correctness of the code on a simpler sequential or nearly-sequential execution model.

A key simplifying assumption available to the developer of sequential code is the determinism of the execution model. In contrast, parallel programs exhibit several sources of nondeterminism. The most obvious one is the nondeterministic behavior due to the interleaving of parallel threads, sometimes called *scheduler nondeterminism*. Scheduler nondeterminism is essential to make parallel threads execute simultaneously and to harness the power of parallel chips. We believe that often the programmer strives to preserve the determinism of his application even in face of nondeterministic scheduling.

In previous work [4] we argued that programmers should be provided with a framework to allow them to *express deterministic behavior of parallel programs directly and easily*. We believe that for many parallel programs the deterministic aspect of the execution can be verified, either through static or through dynamic analysis, without the need to verify much of the functional correctness of the application. In the terms of our earlier discussion about decomposing correctness reasoning, determinism checking ensures the correctness of the parallelization—i.e., the irrelevance of scheduling nondeterminism—largely separated from the functional correctness of the application.

We observe that manually parallelized programs often employ algorithms that are different than their sequential versions. In some cases the algorithm itself is nondeterministic. Examples of such nondeterministic algorithms include branch-and-bound algorithms, which may produce multiple valid outputs for the same input. Such algorithmic nondeterminism is very often tightly coupled with the functional correctness of the code. In these cases, efforts to verify the deterministic behavior of a program become entangled in the program's functional correctness. To address such programs, we propose to separate the reasoning about the algorithmic and the scheduler sources of nondeterminism

Thus, we argue for a specification technique based on *nondeterministic sequential programs*. Using this technique the programmer will have an opportunity to express the expected algorithmic nondeterminism in a sequential program. Reasoning about the correctness of the nondeterministic sequential program is still difficult, but not much more difficult than reasoning about the correctness of the sequential program. In contrast, we expect to be able to develop automatic techniques to show that the scheduler nondeterminism does not add any *additional* behavior to a nondeterministic sequential program.

Using such a specification technique, we expect to be able to map dynamic traces of a parallel execution of the program into equivalent nondeterministic sequential traces where there is no interleaving, although some of constructs are resolved nondeterministically. As a concrete use of these techniques, we envision a parallel-program *debugging methodology* that presents to the programmer a virtual nondeterministic sequential view of a parallel execution. Thus, the programmer debugs the actual parallel execution, but needs to focus only on the correctness of the nondeterministic sequential version.

In the rest of this paper, we first briefly describe our previous work on specifying and checking determinism, and describe an example where the verification of such a specification cannot be separated from the functional correctness of a program (Section 2). We then describe our idea of using nondeterministic sequential specifications on this motivating example (Section 3).

2 Specifying Semantic Determinism

In our previous work [4], we argued that programmers should be provided with a framework that will allow them to *express deterministic behavior of parallel programs directly and easily*. Formally, we proposed [4] the following construct for the specification of determinism:

```
deterministic assume(Pre($s_0, s'_0$)) {
    P
} assert(Post($s, s'$));
```

Here Pre and Post are predicates over two program states in different executions resulting from different thread schedules. Formally, this specification states that for any program states s_0, s'_0, s, and s', if (1) Pre(s_0, s'_0) holds, (2) an execution of P from s_0 terminates and results in state s, and (3) an execution of P from s'_0 terminates and results in state s', then Post(s, s') must hold.

The advantage of our deterministic specifications is that they provide a way to specify the correctness of just the use of parallelism in a program, independent of the full functional correctness. We also developed a directed testing based approach to check such specifications.

2.1 Motivating Example

Consider the generic, sequential branch-and-bound procedure given in Figure 1. The search procedure finds a minimum-cost solution in a given solution space. Initially, the FIFO work-queue queue holds a single element representing the entire space to be searched. The procedure repeatedly retrieves a unit of work—i.e. a region of the solution space— from queue and either: (1) prunes the region if its lower bound exceeds the best cost found so far, (2) exhaustively searches for the minimal solution the region when the region is smaller than some threshold, or (3) splits the regions into smaller pieces for later processing.

Consider the parallel version of this branch-and-bound procedure given in Figure 2. The code is quite similar to that in Figure 1, but the parallel version uses a parallel-for construct which can spawn a new parallel iteration whenever queue contains a work item. The loop terminates when all spawned iterations have finished and queue is empty. Further, the code uses an atomic construct to safely update shared variables best and best_soln in Lines 7-10. (And the queue data structure must be thread-safe.)

2.2 The Challenge of Separating Parallel and Functional Correctness

We would like to verify that the procedure in Figure 2 is a correct parallelization of the sequential procedure in Fig-

```
for (work in queue) {
  if (soln_lower_bound(work) >= best)
    continue;
  if (size(work) < THRESHOLD) {
    soln = find_min_soln(work);

    if (cost(soln) < best) {
      best = cost(soln);
      best_soln = soln;
    }

  } else {
    queue.add(split(work));
  }
}
```

Figure 1: A generic branch-and-bound procedure.

```
par-for (work in queue)) {
  if (soln_lower_bound(work) >= best)
    continue;
  if (size(work) < THRESHOLD) {
    soln = find_min_soln(work);
    atomic {
      if (cost(soln) < best) {
        best = cost(soln);
        best_soln = soln;
      }
    }
  } else {
    queue.add(split(work));
  }
}
```

Figure 2: A *parallel* branch-and-bound procedure.

ure 1. Ideally, this effort would require reasoning about the parallel and synchronization constructs in Figure 2, but would not require us to reason about the functional correctness of the sequential code.

Using our deterministic specification framework, we can express the parallel correctness as follows:

```
deterministic
  assume (queue.equals(queue')) {
    ... code from Figure 2 ...
} assert (best == best');
```

That is, that every run of the parallel search on the same input finds a solution with the same minimal cost. Similarly, we could attempt to show that every parallel execution produces a solution with the same cost as the sequential procedure. We argue that such an effort cannot escape complex reasoning about the functional correctness of the code.

The functional correctness of a branch-and-bound search depends critically on the correctness of the bounding procedure soln_lower_bound. In particular, for any solution s in a region w, we must have that cost(s) $\geq$ soln_lower_bound(w). This guarantees that it is safe to prune region work when we have soln_lower_bound(work) $\geq$ best—because no soln in work can have cost less than best_soln.

Suppose we have an instance of the generic branch-and-bound procedure in which soln_lower_bound contains a bug. In particular, suppose best $= 100$ and queue is as shown in Figure 3.

The sequential procedure happens to find the correct minimal solution in this case. The procedure first processes item a. The procedure will not prune item a because soln_lower_bound(a) < best. After searching for find_min_soln(a), the procedure updates best to 2. Then, region b will similarly not be pruned, but best will not be updated because cost(find_min_soln(b)) > 2. Finally, region c will be pruned because soln_lower_bound(c) > 2. Note that this is also a possible execution of the parallel version of the procedure.

The parallel procedure, even if correctly parallelized, may return an incorrect solution. This procedure can process a, b, and c all in parallel. Suppose it first computes soln_lower_bound(b) < 100 and then computes soln_lower_bound(c) < 100, and thus prunes neither. Suppose further that the procedure then searches regions b and c for find_min_soln(b) and find_min_soln(c), updating shared variable best to cost(find_min_soln(b)) $= 3$. The procedure will then prune a because soln_lower_bound(a) is incorrectly larger than 3.

Thus, we cannot show that the parallel version of the search produces equivalent results to the sequential version. Yet, because queue is a thread-safe concurrent queue and because we ensure that updates to best and best_soln are atomic, we can clearly see that the procedure has been *parallelized* correctly. The error here is essentially a *sequential* bug.

This suggests that we need a weaker notion of *parallel correctness*. To have a hope of statically proving the correctness of the parallelism in such complex programs, even when they are sequentially correct, we need a notion of parallel correctness that is better decoupled from functional correctness.

3 Nondeterministic Specifications

We propose specifying the parallel correctness of a program using a *nondeterministic* but *sequential* version of the program. We argue that such an approach decomposes the effort of verifying a parallel program into: (1) verifying the parallelism by showing that the nondeter-

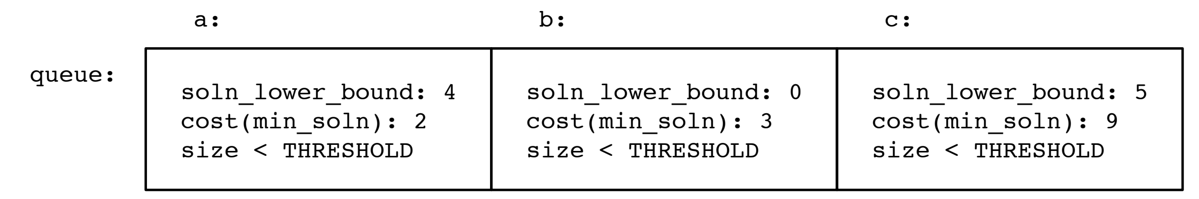

Figure 3: A `queue` from the generic branch-and-bound procedure of Figures 1, 2, and 4. Note that `soln_lower_bound(a)` is not correct—it should be smaller than `cost(find_min_soln(a))`.

ministic sequential and parallel versions of the program are equivalent, and (2) verifying functional correctness of the nondeterministic sequential version.

Consider the branch-and-bound procedure given in Figure 4. This code is the same as the parallel procedure in Figure 2, except that we have added a nondeterministic Boolean value to the condition at Line 2: "**&& ***". That is, even when the procedure finds that it could prune region `work`, we permit it to nondeterministically process `work` anyway. Further, we have replaced the `parallel-for` construct with a nondeterministic `for`-loop. This loop may remove items from `queue` in any order, rather than always removing from the front of `queue`, but is restricted to run sequentially—i.e. one iteration at a time.

We can use this nondeterministic, sequential version of the procedure, in Figure 4, to decompose the verification effort for the parallel code in Figure 2. The first piece of the verification effort is to show that the parallel version is equivalent to the nondeterministic sequential version. We can think of the nondeterministic sequential version as a specification of the parallel version, in which a programmer annotates the *acceptable* or *expected* nondeterminism. In this view, the parallelization is correct when the results of nondeterministic parallel thread scheduling exhibit only this expected nondeterminism.

The second piece of the verification effort—completing the functional correctness of the parallel code—can then be performed on the nondeterministic but sequential version. This step remains challenging. However, since it does not need to consider scheduler nondeterminism it should be only a slight extension of the functional correctness argument of the sequential version. In fact for such correctness verification, one could use model-checking algorithms and tools based on predicate abstraction [16, 2, 12] that have been developed for both deterministic and nondeterministic sequential programs with procedures. These model checkers use the fact that reachability of configurations of pushdown systems is decidable [1, 6]. Note that such techniques cannot be directly applied to verify functional correctness of parallel programs, as the verification of multithreaded Boolean programs with procedures is undecidable [15].

For ease of use by programmers, we propose that a parallel program and its nondeterministic and sequential executable specification can be the same software artifact, with two different interpretations or semantics. That is, nondeterministic expressions (*'s) can be added to the code of the parallel program, but default to always being `true` or `false`, depending on the context, when the code is interpreted as a parallel program. But when the code is interpreted as a nondeterministic sequential program, the expressions are nondeterministically `true` or `false`. Similarly, parallel constructs such as `parallel-for` or `cobegin` are given different semantics under the nondeterministic sequential interpretation—for example, with a `parallel-for` becoming a nondeterministic `for`-loop as in Figures 2 and 4.

3.1 Reduction for Parallel Correctness

We propose using a reduction-based [14] approach to prove parallel correctness by showing the equivalence of a parallel program to its nondeterministic, sequential version. The key idea is to show that, given an execution trace of a parallel program, we can construct a trace

```
 1: nondet-for (work in queue)) {
 2:  if ((soln_lower_bound(work) >= best)
                                    && *)
 3:     continue;
 4:  if (size(work) < THRESHOLD) {
 5:     soln = find_min_soln(work);
 6:     atomic {
 7:       if (cost(soln) < best) {
 8:         best = cost(soln);
 9:         best_soln = soln;
10:       }
11:     }
12:  } else {
13:     queue.add(split(work));
14:  }
15: }
```

Figure 4: A nondeterministic and sequential generic branch-and-bound procedure.

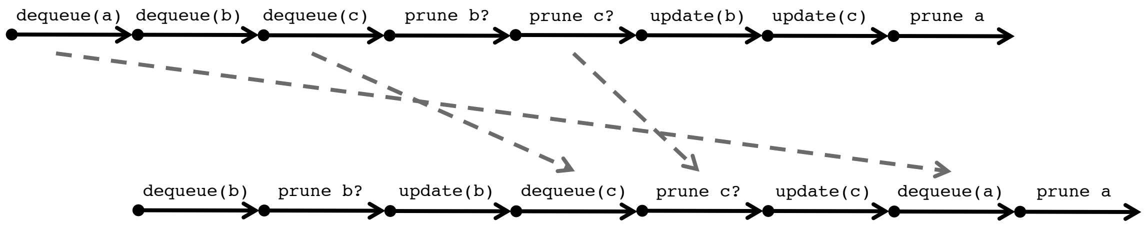

Figure 5: Reduction example. Top trace is from parallel program; bottom is from nondeterministic sequential.

of the nondeterministic sequential program that produces an identical result.

For example, consider the parallel branch-and-bound execution described in Section 2.1. The queue is as shown in Figure 3 and initially best = 100. The top trace in Figure 5 is a trace of this execution: Items a, b, and c are each dequeued in separate threads. Then soln_lower_bound is computed for b and for c and neither is pruned. Then find_min_soln(b) is executed and best is updated with cost(find_min_soln(b)) = 3 (operation "update(b)"). Then the procedure executes find_min_soln(c), and best is not updated because cost(find_min_soln(b)) > best (operation "update(c)"). Finally, soln_lower_bound(a) is computed and a is pruned.

For this execution trace, we can construct an equivalent execution trace of the nondeterministic sequential branch-and-bound procedure, shown as the bottom trace in Figure 5. This is a trace of the nondeterministic sequential procedure—the nondeterministic-for loop processes b, then c, and then a. The dotted arrows in Figure 5 show that this trace is a rearrangement of the parallel trace.

This rearrangement in Figure 5 is a dynamic *reduction* [14] of the execution, guaranteeing that the rearranged execution produces the same result as the original. The reduction is possible because:

- A dequeue operation is a *right-mover*. We can move a dequeue later in an execution relative to the operations of other threads—i.e. move it to the *right*—without changing the result of the program.

- With our added nondeterminism, a prune operation is also a *right-mover*. Without this nondeterminism, we cannot safely move a prune operation later in an execution, relative to the other threads, because some update operation could decrease best. Thus, the prune check would fail when executed before this decrease of best, but succeed if executed after. (On the other hand, if the prune already succeeded, then it would still succeed if moved later in the execution because best never increases.)

  But the added nondeterminism allows the prune operation to decline to prune a region work, even when soln_lower_bound exceeds best. Thus, we can move rightward a failing prune check without changing its effect on the program.

Not shown in the above example are *left-movers*, operations that can be scheduled earlier in the program execution without changing their effect on the program, and *both-movers*, which are both left- and right-movers.

The above example shows that we can use reduction to dynamically check parallel correctness. That is, given a trace of a parallel program, we can classify the operations in the trace as left- and right-movers and report that the parallelism in the trace is correct if, by reordering the left- and right-movers, we can construct a trace of the corresponding nondeterministic sequential program.

In general, we believe that we can statically show, as in [5], that atomic operations in parallel programs, with the added nondeterminism from their nondeterministic sequential specifications, are left- and right-movers. By showing, for example, that each iteration of a parallel loop consists of a series of right-movers, followed possibly by one atomic non-mover, and then a series of left-movers, we can prove that each iteration is *atomic*, and thus the parallel loop is equivalent to its nondeterministic sequential counterpart.

We expect that we may encounter cases when this technique does not work as easily as described here—it could be that the only nondeterministic sequential execution that produces identical output cannot be constructed be rearranging the given parallel execution. In our future work, we plan to investigate the extent to which this technique is applicable to a wide range of parallel programs and, based on experimental evidence, to propose extensions that can increase its applicability.

Acknowledgments

This work supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by Sun Microsystems and by matching funding from UC MICRO (Award #08-113), by NSF Grants CNS-0720906 and CCF-0747390, and by a DoD NDSEG Graduate Fellowship.

References

[1] AUTEBERT, J.-M., BERSTEL, J., AND BOASSON, L. Context-free languages and pushdown automata. 111–174.

[2] BALL, T., AND RAJAMANI, S. The SLAM Toolkit. In *13th Conference on Computer Aided Verification (CAV)* (2001), vol. 2102 of *LNCS*, pp. 260–264.

[3] BOCCHINO, R., ADVE, V., ADVE, S., AND SNIR, M. Parallel programming must be deterministic by default. In *First USENIX Workship on Hot Topics in Parallelism (HOTPAR 2009)* (March 2009).

[4] BURNIM, J., AND SEN, K. Asserting and checking determinism for multithreaded programs. In *7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (2009), ACM.

[5] ELMAS, T., QADEER, S., AND TASIRAN, S. A calculus of atomic actions. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), ACM, pp. 2–15.

[6] FINKEL, A., WILLEMS, B., AND WOLPER, P. A direct symbolic approach to model checking pushdown systems. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)* (1997), vol. 9 of *Electronic Notes in Theor. Comp. Sci.*, Elsevier.

[7] FLANAGAN, C. Verifying commit-atomicity using model-checking. In *11th International SPIN Workshop* (2004), pp. 252–266.

[8] FLANAGAN, C., AND FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2004), pp. 256–267.

[9] FLANAGAN, C., AND QADEER, S. A type and effect system for atomicity. In *Proc. of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'03)* (2003), pp. 338–349.

[10] FREUND, S. N., AND QADEER, S. Checking concise specifications for multithreaded software. *Journal of Object Technology 3*, 6 (2004), 81–101.

[11] HATCLIFF, J., ROBBY, AND DWYER, M. B. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)* (2004), pp. 175–190.

[12] HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy Abstraction. In *29th ACM Symposium on Principles of Programming Languages (POPL)* (2002), pp. 58–70.

[13] KULKARNI, M., PINGALI, K., WALTER, B., RAMANARAYANAN, G., BALA, K., AND CHEW, L. P. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), ACM, pp. 211–222.

[14] LIPTON, R. J. Reduction: A method of proving properties of parallel programs. *Communications of the ACM 18*, 12 (1975), 717–721.

[15] RAMALINGAM, G. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst. 22*, 2 (2000), 416–430.

[16] S. GRAF, AND H. SAIDI. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification (CAV'97)* (1997), vol. 1254 of *LNCS*, pp. 72–83.

[17] WANG, L., AND STOLLER, S. D. Run-time analysis for atomicity. In *3rd Workshop on Run-time Verification (RV'03)* (2003), vol. 89 of *ENTCS*.

[18] WANG, L., AND STOLLER, S. D. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Mar. 2006), ACM Press, pp. 137–146.