NDetermin: Inferring Nondeterministic Sequential Specifications for Parallelism Correctness

Jacob Burnim Tayfun Elmas George Necula Koushik Sen
EECS Department, University of California, Berkeley, CA, USA
{jburnim,elmas,necula,ksen}@cs.berkeley.edu

Abstract

Nondeterministic Sequential (NDSeq) specifications have been proposed as a means for separating the testing, debugging, and verifying of a program's parallelism correctness and its sequential functional correctness. In this work, we present a technique that, given a few representative executions of a parallel program, combines dynamic data flow analysis and Minimum-Cost Boolean Satisfiability (MinCostSAT) solving for automatically inferring a likely NDSeq specification for the parallel program. For a number of Java benchmarks, our tool NDETERMIN infers equivalent or stronger NDSeq specifications than those previously written manually.

1. Introduction

As multicore and manycore processors become increasingly common, more and more programmers must write parallel software. But writing such parallel software can be difficult and error prone. In addition to reasoning about the often-sequential functional correctness of each component of a program in isolation, a programmer must simultaneously consider whether multiple components running in parallel, their threads interleaving nondeterministically, can harmfully interfere with one another.

In an earlier paper [2] we proposed nondeterministic sequential (NDSeq) specifications as a means for decomposing the reasoning about a program's parallelism and its functional correctness. To explain the problem addressed by NDSeq specifications, consider the simple parallel program in Figure 1(a). The program consists of a parallel for-loop, written as `coforeach`—each iteration of this loop attempts to perform a computation (Line 6) on shared variable `x`, which is initially 0. Each iteration uses an atomic compare-and-swap (`CAS`) operation to update shared variable `x`. If multiple iterations try to concurrently update `x`, some of these `CAS`'s will fail and those parallel loop iterations will recompute their updates to `x` and try again.

If we can specify the full functional correctness of our example program—i.e., specify precisely which final values of `x` are correct for each input value of `x`—then this specification will clearly imply that the parallelization of the program was correct. Although it may look straightforward to write such a formal specification for our motivating example, we believe that it will be a very difficult task for many large and complex programs.

```
1: coforeach (i in 1,...,N) {
2:   bool done = false;
3:   while (!done) {
4:
5:     int prev=x;
6:     int curr=i * prev + i;
7:     bool c=CAS(x,prev,curr);
8:     if (c) {
9:       done = true;
10:} } }
```

(a) Parallel program

```
1: nd-foreach(i in 1,...,N) {
2:   bool done = false;
3:   while (!done) {
4:     if (*) {
5:       int prev=x;
6:       int curr=i * prev + i;
7:       bool c=CAS(x,prev,curr);
8:       if (c) {
9:         done = true;
10:} } } }
```

(b) NDSeq specification

Figure 1. Simple parallel program to perform the reduction in line 6 for the integers $\{1,\ldots,N\}$, in some arbitrary order, and an NDSeq specification for the program.

A natural approach to specifying parallelism correctness would be to specify that the program in Figure 1(a) must produce the same final value for `x` as a version of the program with all parallelism removed—i.e., the entire code is executed by a single thread. However, in this case we do not get a sequential program equivalent to the parallel program. For example, the parallel program in Figure 1(a) is free to execute the computations at line 6 in any *nondeterministic* order. Thus, for the same input value of `x`, different thread schedules can produce different values for `x` at the end of the execution. On the other hand, executing the loop sequentially from 1 to `N` will always produce the same, deterministic final value for `x`. Suppose that such extra nondeterministic behaviors due to thread interleavings are intended; the challenge here is how to express these nondeterministic behaviors in a sequential specification.

We addressed this challenge in [2] by introducing a specification mechanism that the programmer can use to declare the intended, algorithmic notions of nondeterminism in the form of a sequential program. Such a *nondeterministic sequential specification* (NDSeq) for our example program is shown in Figure 1(b). This specification is intentionally very close to the actual parallel program, but its semantics is sequential with two nondeterministic aspects. First, the **nd-foreach** keyword in line 1 specifies that the loop iterations can run in any permutation of the set 1, ..., N. This part of the specification captures the *intended* (or *algorithmic*) nondeterminism in the behavior of the program, caused in the parallel program by running threads with arbitrary schedules. Any *additional* nondeterminism is an error, due to unintended interference between interleaved parallel threads, such as data races or atomicity violations. Second, the **`if(*)`** keyword in line 4 specifies that the iteration body may be skipped nondeterministically, at least from a partial correctness point of view; this is acceptable, since the loop in this program fragment is already prepared to deal with the case when the effects of an iteration are ignored following a failed CAS statement. In summary, all the final values of `x` output by the parallel program in Figure 1(a) can be produced by a feasible execution of the NDSeq specification in Figure 1(b). Then, we say that the parallel program obeys its NDSeq specification, and, the functional correctness of a parallel program can be tested, debugged, and verified *sequentially* on the NDSeq specification, without any need to reason about the uncontrolled interleaving of parallel threads.

In [2], we also proposed a sound runtime technique that checks a given representative interleaved execution trace of a structured-parallel program, whether there exists an equivalent, feasible execution of the NDSeq specification. This technique was able to check the parallelism correctness of a number of complex Java benchmark programs.

2. Inferring NDSeq Specifications

The key difficultly with the manual approach is that writing such specifications, and especially the placement of the **`if(*)`** constructs, can be can be a time-consuming and challenging process, especially to a programmer unfamiliar with such specifications. If a programmer places too few **`if(*)`** constructs, she may not be able to specify some intended nondeterministic behaviors in the parallel code. However, if she places too many **`if(*)`** constructs, or if she place them in the wrong places, the specification might allow too much nondeterminism, which will likely violate the intended functionality of the code.

Therefore, we believe that automatically inferring NDSeq specifications can save programmer time and effort in applying NDSeq specifications. In particular, we believe that using an inferred specification as a starting point is much simpler than writing the whole specification from scratch. Further, our inference algorithm can detect parallel behaviors that *no* possible NDSeq specification would allow, which often contain parallelism bugs. More generally, such inferred specifications can aid in understanding and documenting a program's parallel behavior. Finally, inferring NDSeq specifications is a step towards an automated approach to testing and verification of parallel programs by decomposing parallelism and sequential functional correctness.

Our contribution in this work is to give an algorithm, running on a set of input execution traces, for inferring a *minimal* nondeterministic sequential specification such that the checking approach described in [2] on the input traces succeeds. Choosing a minimal specification—i.e., with a minimal number of **`if(*)`**, is a heuristic that makes it more likely that the inferred specification matches the intended behavior of the program.

Our key idea is to reformulate the runtime checking algorithm in [2] (explained below) as a constraint solving and optimization problem, in particular a Minimum Cost Boolean Satisfiability (MinCostSAT) problem.

Given a parallel execution of such a program, our algorithm in [2] performs a *conflict-serializability* [4] check to verify that the same behavior could have been produced by the NDSeq version of the program. For our example program in Figure 1, we think of each parallel loop iteration as a transaction and check an interleaved execution of the parallel loop can be serialized with respect to the NDSeq specification—i.e. whether the final result (for our example, the value of the shared variable `x`) can be obtained by running the loop iterations sequentially in some nondeterministic order. But first, our technique combines a dynamic dependence analysis with a program's specified nondeterminism to show that conflicts involving certain operations in the trace can be soundly ignored when performing the conflict-serializability check.

In order to report an execution serializable, we must be able to show that all conflict cycles between parallel transactions can be safely ignored. For this, we perform a dynamic data flow analysis and use the **`if(*)`** in the program's NDSeq specification in this analysis. In particular, we need to identify *relevant* events in the traces: (i) final writes to the shared variable `x`, and (ii) all events on which events in (i) are (transitively) *dependent*. Then, we check if there is any conflict cycle formed by only relevant events; we can safely ignore the cycles that contain irrelevant events.

Benchmark		# Parallel Constructs	# `if(*)`'s	Inferred NDSeq Specification	
				# `if(*)`'s	Correct?
JGF	`sor`	1	0	0	yes
	`matmult`	1	0	0	yes
	`series`	1	0	0	yes
	`crypt`	2	0	0	yes
	`moldyn`	4	0	0	yes
	`lufact`	1	0	0	yes
	`raytracer`	1	0	-	-
	`raytracer (fixed)`	1	0	0	yes
	`montecarlo`	1	0	0	yes
PJ	`pi3`	1	0	0	yes
	`keysearch3`	2	0	0	yes
	`mandelbrot`	1	0	0	yes
	`phylogeny`	2	3	-	-
	`phylogeny (fixed)`	2	3	1	yes
`non-blocking stack`		1	2	2	yes
`non-blocking queue`		1	2	2	yes
`meshrefine`		1	2	2	yes

Table 1. Experimental results. All **`if(*)`** annotations inferred by our tool were verified manually to be correct.

In order to infer an NDSeq specification, we observe a set of representative parallel execution traces for which the standard conflict serializability check gives conflict cycles. Since we are inferring an NDSeq specification for the program, not for a single trace, using multiple traces allows us to observe variations in the executions and improves the reliability of the inferred NDSeq specification.

We then construct and solve a MinCostSAT formula that takes as input the events in the input traces and the conflict cycles detected by the standard conflict serializability check. While generating the formula, we encode the reasoning about relevant events and conflict cycles described above as constraints in the formula. In particular, the constraints enforce the data dependencies between the events and conditions to ignore all observed conflict cycles in the input traces. The MinCostSAT formulation contains variables corresponding to possible placement of **`if(*)`**s in the program. If this formula is satisfiable, then the solution gives us a minimal set of statements $\mathcal{S}*$ in the program, such that the input traces are all serializable with respect to the NDSeq specification obtained by enclosing all statements in $\mathcal{S}*$ with **`if(*)`**. The minimal such solution for our example in Figure 1(a) places a single **`if(*)`** that encloses lines 5-10. Thus, our algorithm produces the correct NDSeq specification given in Figure 1(b). We refer to the reader to our technical report [1] for the details of our MinCostSAT formulation.

3. Results

In this section, we describe our efforts to experimentally evaluate our approach to inferring likely nondeterministic sequential (NDSeq) specifications for parallel programs. In particular, we aim to evaluate the following claim: By examining a small number of representative executions, our specification inference algorithm can automatically generate the correct set of **`if(*)`** annotations for real Java programs.

To evaluate this claim, we implemented our technique in a prototype tool for Java, called NDETERMIN, and applied NDETERMIN tool to the set of Java benchmarks for which NDSeq specifications were previously written manually [2]. We compared the quality and accuracy of our automatically-inferred **`if(*)`**s to the ones in their manually-written NDSeq specifications.

The names, sizes, and brief descriptions of the benchmarks we used to evaluate NDETERMIN are listed in Table 1. Several benchmarks are from the Java Grande Forum (JGF) benchmark suite [5] and the Parallel Java (PJ) Library [3].

The results of our experimental evaluation are summarized in Table 1. (See our technical report [1] for the full table.) The column labeled "All", under "Size of Trace (Events)", reports the number of total events seen in the last execution (of five) of each benchmark, and the column labeled "Sliced Out" reports the number of events removed by our dynamic slicing. NDETERMIN searches for **if(*)** placements to eliminate cycles of transactional conflicts involving sliced out events.

The second-to-last column of Table 1 reports the number of **if(*)** constructs in the inferred NDSeq specification for each benchmark. We manually determined whether each of the inferred **if(*)** annotations was correct—i.e., captures all intended nondeterminism, so that the parallel program is equivalent to its NDSeq specification, but no extraneous nondeterminism that would allow the NDSeq version of the program to produce functionally incorrect results. All of the inferred specifications were correct.

For many of the benchmarks, NDETERMIN correctly infers that no **if(*)** constructs are necessary. All but one of these benchmarks are simply conflict-serializable. As discussed in [2], montecarlo is not conflict-serializable, but the non-serializable conflicts afftect neither the control-flow nor the final result of the program.

For benchmarks stack, queue, and meshrefine, NDETERMIN infers an NDSeq specification exactly equivalent to the manual specifications from [2]. That is, NDETERMIN infers the same number of **if(*)** constructs and places them in the same locations as in previous manually-written NDSeq specifications. We note that NDETERMIN finds specifications slightly smaller than the manual ones, which include a small number of adjacent statements in the **if(*)** that do not strictly need to be enclosed, although in each case the overall behavior of the NDSeq specification is the same whether or not these statements are included in the **if(*)**.

Further, for benchmark phylogeny (fixed), while the previous manual NDSeq specification included three **if(*)** constructs, NDETERMIN correctly infers that only one of these three is actually necessary. The extra **if(*)** appear to have been manually added to address some possible parallel conflicts that, in fact, can never be involved in non-serializable conflict *cycles*. That is, these two extraneous **if(*)** allow the NDSeq specification to perform several nondeterministic behaviors seen during parallel execution of the benchmark. But NDETERMIN correctly determines that these behaviors are possible in the NDSeq specification even without these **if(*)**.

Note that for two benchmarks, raytracer and phylogeny, NDETERMIN correctly reports that no NDSeq specification (i.e., no solution to the SAT instance) exists (indicated by "-" in Table 1). That is, NDETERMIN detects that the events of the dynamic slice (i.e., those not removed by dynamic slicing) are not conflict-serializable. These conflicts exist because both benchmarks contain parallelism errors (atomicity errors due to insufficient synchronization). As a result of these errors, these two parallel applications can produce incorrect results that no sequential version could produce.

These experimental results provide promising preliminary evidence for our claim that NDETERMIN can automatically check serializability by way of inferring **if(*)** necessary for the NDSeq specification of parallel correctness for real parallel Java programs. We believe adding nondeterministic **if(*)** constructs is the most difficult piece of writing a NDSeq specification, and thus our inference technique can make using NDSeq specifications much easier. Further, such specification inference may allow for fully-automated testing and verification to use NDSeq specifications to separately address parallel and functional correctness.

References

[1] NDetermin: Inferring nondeterministic sequential specifications for parallelism correctness. Technical report. *http://goo.gl/W3RM8*.

[2] J. Burnim, T. Elmas, G. Necula, and K. Sen. NDSeq: Runtime checking for nondeterministic sequential specifications of parallel correctness. In *Programming Language Design and Implementation (PLDI)*, 2011.

[3] A. Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[4] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[5] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Supercomputing (SC)*, 2001.