NDetermin: Inferring Nondeterministic Sequential Specifications for Parallelism Correctness

Jacob Burnim Tayfun Elmas George Necula Koushik Sen

Department of Electrical Engineering and Computer Sciences, University of California, Berkeley

{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.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Reliability, Verification

1. Nondeterministic Sequential Specifications

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 correctness 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.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

```
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:     if (CAS(x,prev,curr)) {
8:       done = true;
9: } } }
```

(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:       if (CAS(x,prev,curr)) {
8:         done = true;
9: } } } }
```

(b) NDSeq specification

Figure 1. Example program to perform the reduction in line 6 for the integers $\{1,\ldots,N\}$, in some arbitrary order.

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.

2. Inferring NDSeq Specifications

The key difficulty 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. More generally, such inferred specifications can aid in understanding and documenting a program's parallel behavior.

In [2], we proposed a sound runtime technique that, using conflict-serializability check, checks a given representative interleaved execution trace of a structured-parallel program, whether there exists an equivalent, feasible execution of the NDSeq specification. 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] as a constraint solving and optimization problem, in particular a Minimum Cost Boolean Satisfiability (MinCostSAT) problem. 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 indicating possible violations of the NDSeq specification. Similarly to the algorithm in [2], we utilize a dynamic dependence analysis with a program's specified nondeterminism in our MinCostSAT formulation. The constructed MinCostSAT formula 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—i.e., conflict cycles involving these statements in the input traces can be soundly ignored—with respect to the NDSeq specification obtained by enclosing each statement in $\mathcal{S}$ with `if(*)`. The minimal such solution for our example in Figure 1(a) places a single `if(*)` that encloses lines 5-8. Thus, our algorithm produces the correct NDSeq specification given in Figure 1(b). We refer the reader to our technical report [1] for the further details of our MinCostSAT formulation.

3. Results

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

Our experimental results are summarized in Table 1. The second-to-last column of Table 1 reports the number of `if(*)` constructs in the inferred NDSeq specification for each benchmark. We manually confirmed that each of the inferred `if(*)` annotation was correct. For many of the benchmarks, NDETERMIN correctly inferred that no `if(*)` constructs are necessary.

Benchmark		# Parallel Constructs	# if(*)'s in manual specification	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.

For benchmarks `stack`, `queue`, and `meshrefine`, we note that NDETERMIN finds specifications slightly smaller than the manual ones. 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(*)`'s appear to have been manually added to address some possible parallel conflicts that, in fact, can never be involved in a non-serializable execution. Finally, our inference algorithm can detect parallel behaviors that *no* possible NDSeq specification would allow, which often contain parallelism bugs. In fact, as indicated by "–" in Table 1, NDETERMIN correctly refuses to infer NDSeq specifications for the buggy versions of `raytracer` and `phylogeny` (containing atomicity errors due to insufficient synchronization), since no solution to the MinCostSAT instance exists; NDETERMIN does infer correct NDSeq specifications for the correct (`fixed`) versions of these benchmarks. (For a more elaborate discussion of our experimental results, see our technical report [1].)

These experimental results provide promising preliminary evidence for our claim that NDETERMIN can automatically infer `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.

Acknowledgments

This research supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), by NSF Grants CCF-101781, CCF-0747390, CCF-1018729, and CCF-1018730, and by a DoD NDSEG Graduate Fellowship. The last author is supported in part by a Sloan Foundation Fellowship. Additional support comes from Oracle (formerly Sun Microsystems), from a gift from Intel, and from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, and Samsung.

References

[1] J. Burnim, T. Elmas, G. Necula, and K. Sen. NDetermin: Inferring nondeterministic sequential specifications for parallelism correctness. Technical Report UCB/EECS-2011-143, EECS Department, University of California, Berkeley, Dec 2011.

[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.