NDetermin: Inferring Nondeterministic Sequential Specifications for Parallelism Correctness

Jacob Burnim
Tayfun Elmas
George Necula
Koushik Sen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-143
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-143.html

December 16, 2011

Acknowledgement

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 CNS-0720906, 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.

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

A key reason for the great difficulty of writing, testing, and verifying parallel programs is the need to reason simultaneously about not only the sequential correctness of each part of a program in isolation, but also about all possible nondeterministic interleavings of the program's parallel threads. Thus, there has been much interest in techniques for separately testing or verifying the correctness of a program's use of parallelism, allowing the program's functional correctness to be tested or verified in a sequential way.

Nondeterministic Sequential (NDSeq) specifications have been proposed as a means for achieving this decomposition in testing, debugging, and verifying a program's parallelism correctness and its sequential functional correctness. An NDSeq specification for a parallel program is a sequential version of the program, with no parallel threads but with some limited, controlled nondeterminism. A program's use of parallelism is correct if, for every execution of the parallel program, there exists an execution of the NDSeq specification producing the same result. Functional correctness can then be checked on this NDSeq specification, without any interleaving of parallel threads.

While NDSeq specifications have been used successfully to check parallelism correctness, manually writing NDSeq specifications for programs can be a challenging and time-consuming process. Thus, in this work, we propose a technique for automatically inferring a likely NDSeq specification for a parallel program. Given a few representative executions of a parallel program, our technique combines dynamic data flow analysis and Minimum-Cost Boolean Satisfiability (MinCostSAT) solving to infer a likely NDSeq specification for the program's parallelism. We have implemented our technique for Java in a prototype tool called NDETERMIN. For a number of benchmarks, our tool 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 sequential 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. The need to deal simultaneously with the correctness of a program's parallelism and its often-sequential functional correctness greatly complicates writing, testing, debugging, and verifying parallel programs.

We have previously proposed *nondeterministic sequential (NDSeq) specifications* [4] as a way to separately address the correctness of a program's parallelism and a program's sequential functional correctness. The key idea is to use a sequential version of a structured-parallel program as a specification for the program's parallel behavior. That is, the program's use of parallelism is correct if, for every parallel execution of the program, their exists an execution of the sequential version that produces the same result. But, to capture all intended behaviors of a parallel program, the sequential version may have to include some limited, controlled nondeterminism. Thus, a programmer specifies the *intended* or *algorithmic* nondeterminism in their parallel application, and the nondeterministic sequential (NDSeq) specification is a sequential version of the program, with no interleaving of parallel threads but with this annotated nondeterminism. Any *additional* nondeterminism is an error, due to unintended interference between interleaved parallel threads, such as data races or atomicity violations. Further, the functional correctness of a parallel program can then be tested, debugged, and verified *sequentially* on the NDSeq specification, without any need to reason about the uncontrolled interleaving of parallel threads.

In [4], we also proposed a sound runtime technique for checking that a structured-parallel program conforms to its NDSeq specification. This technique was able to check the parallelism correctness of a number of complex Java benchmark programs. However, writing NDSeq specifications for parallel programs can be a time-consuming and challenging process, especially to a programmer unfamiliar with such specifications. A programmer can easily forget to include some correct nondeterministic behaviors, forcing them to iterate between checking their program against its NDSeq specification and inspecting violating executions in order to add such missing nondeterminism to the specification.

In this work, we propose a technique for automatically inferring a likely NDSeq for a structured-parallel program. Given a representative set of correct parallel executions, plus some simple annotations about which program locations contain the final program result, our algorithm infers a NDSeq specification with a minimal amount of added nondeterminism necessary to capture all behavior seen in the observed parallel executions. Our inference algorithm combines dynamic data-flow analysis, conflict-serializability checking, and Minimum-Cost Boolean Satisfiability (MinCostSAT) solving.

We have implemented our NDSeq specification inference algorithm in a prototype tool for Java, called NDETERMIN. We applied NDETERMIN to the set of Java benchmarks for which we previously hand-wrote NDSeq specifications for the work presented in [4], and NDETERMIN correctly inferred all these hand-written specifications. This provides promising preliminary evidence that NDETERMIN can infer correct and useful NDSeq specifications for parallel applications.

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

```
 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:          }
11:  }   }
```

Figure 1. Simple parallel program to perform the reduction in line 6 for the integers {1,...,N}, in some arbitrary order.

```
 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:              }
11:  }   }   }
```

Figure 2. A nondeterministic sequential specification for the program in Figure 1.

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. In such an approach, a program's parallelism would be checked against its inferred NDSeq specification, while functional correctness would be checked *sequentially* on the NDSeq specification using any of a wide variety of powerful techniques for testing and verifying sequential, nondeterministic programs.

2. Overview

In this section, we give an overview of our algorithm for inferring NDSeq specifications for parallel programs on a simple example. We first present some background on NDSeq specifications.

2.1 Motivating Example

Consider the simple parallel program in Figure 1. 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.

Consider the parallel execution shown in Figure 3. In this execution, the `i=1` iteration reads and computes an updated value for shared variable `x`. But before the `i=1` iteration can update `x`, the `i=2` iteration (in another thread) runs and sets `x` to 2. The first compare-and-swap (`CAS`) operation in the `i=1` iteration then fails, and the iteration redoes its computation before successfully updating `x`.

2.2 Background: Nondeterministic Sequential Specifications

The problem addressed by nondeterministic sequential (NDSeq) specifications is how to reason about the correctness of the parallel program in the presence of thread interleavings. If we can specify the full functional correctness of a program—i.e., for our motivating example, 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.

Instead of testing or verifying a parallel program directly against a functional specification, we would like to separate this end-to-end reasoning into two simpler tasks: (1) the checking of whether the program is *parallelized correctly* independent of the complex functional correctness and (2) the checking of whether the program satisfies a functional correctness criteria independent of any interleaving of threads.

A natural approach to specifying parallelism correctness would be to specify that the program in Figure 1 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. (Note that, this condition is independent of which final values of `x` are correct, which is specified and checked separately as the functional correctness of the program.) However, in this case we do not get a sequential program equivalent to the parallel program. For example, the parallel program in Figure 1 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 [4] 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 2. 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 nondeterministic behavior of the program, caused in the parallel program by running threads with arbitrary schedules. 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 can be produced by a feasible execution of the NDSeq specification in Figure 2. Then, we say that the parallel program obeys its NDSeq specification. In fact, we have developed a sound algorithm [4] that checks for a given representative interleaved execution trace τ of the parallel program, whether there exists an equivalent, feasible execution of the NDSeq specification.

2.3 Inferring NDSeq Specifications

The key difficultly with the previous approach is that writing such specifications, and especially the placement of the **`if(*)`** constructs, can be difficult in many practical situations. If we place too few **`if(*)`** constructs, we may not be able to specify some intended nondeterministic behaviors in the parallel code. However,

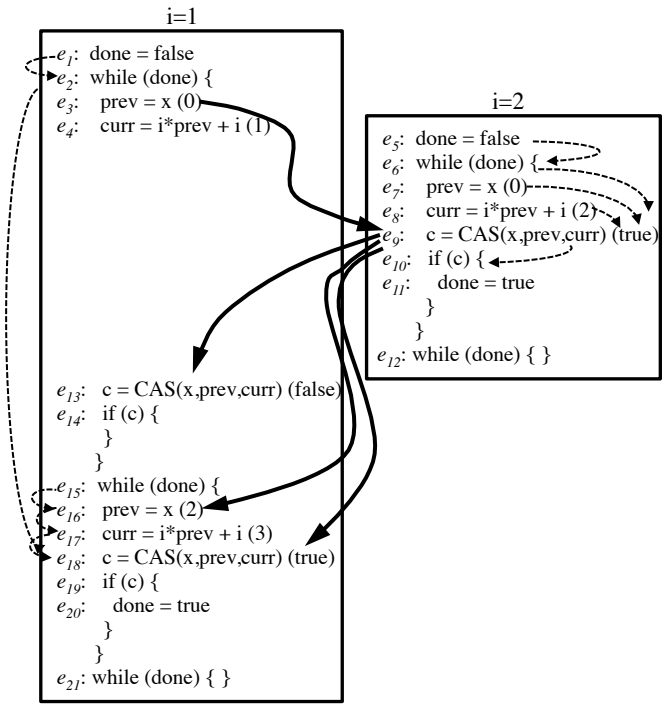

Figure 3. A parallel execution of two iterations (i=1,2) of the example parallel program from Figure 1. The vertical order of events shows the interleaving. Each assignment shows in parentheses the value being assigned. The thin dotted arrows denote data dependencies between events. The thick solid arrows denote transactional conflicts.

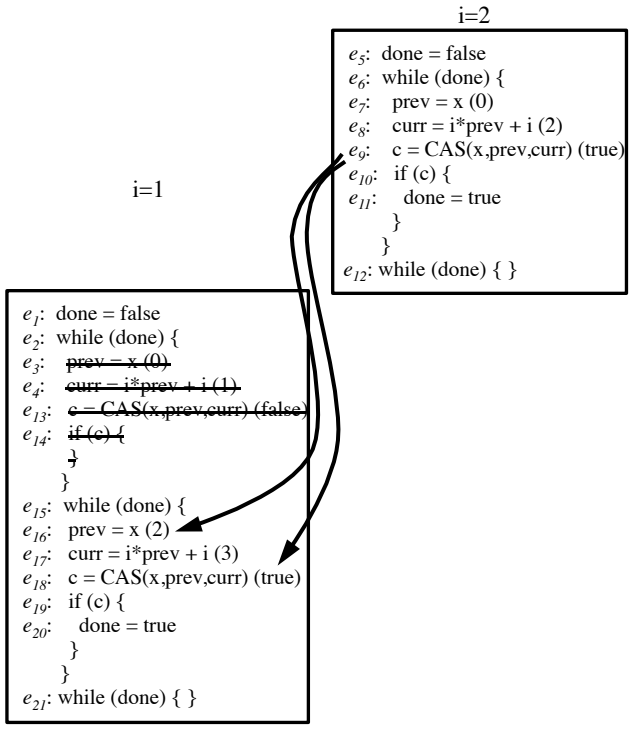

Figure 4. A serialization of the parallel execution in Figure 3. This serialization becomes a witness that there exists an equivalent execution of the NDSeq specification in Figure 2. The vertical order of events shows the interleaving. Each assignment shows in parentheses the value being assigned. The thick solid arrows denote transactional conflicts.

if we place too many `if(*)` constructs, or if we place them in the wrong places, the specification might allow too much nondeterminism, which will likely violate the intended functionality of the code.

Our contribution in this paper 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 [4] 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 our runtime checking algorithm in [4] (explained below) as a constraint solving and optimization problem, in particular a Minimum Cost Boolean Satisfiability (MinCostSAT) problem.

Runtime Checking Parallelism Correctness Consider the parallel execution shown in Figure 3. Our algorithm in [4] checks if this trace can be serialized with respect to the NDSeq specification—i.e. whether the final result (the value of the shared variable `x`) can be obtained by running the loop iterations sequentially in some nondeterministic order. For our example trace, the algorithm discovers the serialization in Figure 4. This serialization is a *witness* to the correctness of the parallelism in the trace in Figure 3.

The algorithm in [4] determines whether or not such an NDSeq execution exists by generalizing *conflict-serializability* [27] checking. We now describe conflict-serializability on our motivating example to show why and how conflict-serializability must be generalized for checking NDSeq specifications. Given a collection of transactions—in this case, we think of each parallel loop iteration as a transaction—we form the *conflict graph* whose vertices are the transactions and with a *conflict edge* from transaction tr to tr' if tr and tr' contain conflicting operations *op* and *op*$'$ with *op* happening before *op*$'$. Two operations from different threads are *conflicting* if they operate on the same shared global and at least one of them is a write; in Figure 3 and 4 the conflicts are shown with thick solid arrows. It is a well-known result [27] that if there are no cycles in the conflict graph, then the transactions are serializable.

Because the conflict arrows from the `i=1` iteration to the `i=2` iteration (from e_3 to e_9) and vice versa (e.g., from e_9 to e_{13}, e_{16}, or e_{18}) form a cycle, these two iterations are not *conflict-serializable*. Yet, this execution trace is serializable, since its result is the same as if we run first the iteration `i=2` followed by `i=1`. Therefore, we need a more general notion than conflict-serializability.

In order to report this execution serializable, we must be able to show that all conflict cycles between iteration `i=1` and `i=2` 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: (1) final writes to the shared variable `x`, and (2) all events on which events in (1) 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.

Computing Relevant Events When computing the set of relevant events, we consider all data dependencies between events (shown with thin dotted arrows in Figure 3). For the trace in Figure 3, we first include events e_9 and e_{18} in the relevant events, as both write to shared variable `x`. We then include e_7, e_8, e_{16}, and e_{17}, as e_9 and e_{18} are *data dependent* on these events.

The way we consider control dependencies is subtle. By default, a deterministic branch event is considered relevant and all events that flow into its branch condition become relevant. For example, in Figure 3, the events e_2, e_6, e_{10}, e_{12}, e_{19}, and e_{21} are considered relevant, and we include events e_1 and e_5, as e_2 and e_6 are data dependent on the writes of local variable `done`. Exceptionally, a branch event can be considered irrelevant if that event is executed by a statement s enclosed within **`if(*)`** in the program's NDSeq specification and all the events generated by that execution of s are irrelevant. Intuitively, this means that in the corresponding execution of the NDSeq specification, that particular execution of s can be entirely ignored without affecting the final outcome of the execution (by considering that the nondeterministic **`if(*)`** will be resolved to **`if`**(**`false`**) in the corresponding NDSeq execution). Therefore, when inferring the NDSeq specification, we need to look for statements to add **`if(*)`** so that we can ignore events that are involved in conflict cycles. In the presence of the data dependencies between events, this becomes a combinatorial search problem.

In order to show that the execution Figure 3 is serializable, we need to ignore the conflict cycles formed by the thick solid arrows in the figure. For this, possible candidate events to ignore are: (1) the read e_3, (2) the write e_9, or (3) all three of e_{13}, e_{16}, and e_{18}. But, since the events e_9, e_{16}, and e_{18} affect the computation of **`x`**, they are relevant for the final result of the trace and they could not be eliminated in a matching serialization even if they were guarded by **`if(*)`**. Thus, our inference algorithm must focus on placing **`if(*)`** around events e_3 and e_{13}.

If we enclose an **`if(*)`** around lines 5-10 as shown in Figure 2, we can safely mark event e_{14} irrelevant, because the branch e_{14} corresponds to is not evaluated, and thus, does not affect the rest of the execution. This also makes the events e_3, e_4, and e_{13} irrelevant because these events flow into only each other and e_{14}. Therefore, we can ignore the events e_3, e_4, e_{13}, and e_{14} together with the conflict cycles they are involved in. In fact, the only conflict cycles in the execution are formed by the events e_3 and e_{13}, and after ignoring these cycles, we can declare the execution in Figure 3 serializable. Serializing this execution respecting the remaining conflict edges gives us the execution trace in Figure 4. This trace can also be generated by a nondeterministic execution of the NDSeq specification given in Figure 2 by choosing *false* for **`if(*)`** in the first iteration of `i=1`.

MinCostSAT Solving for `if(*)` Placements Having explained how a given parallel execution trace is checked against an existing NDSeq specification, we next explain how to infer such a specification. For this, 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(*)`**.

Constraints are added to impose a number of conditions:

1. For each cycle of transactional conflicts, at least one of the events involved in the cycle must be made irrelevant. For example, we would add constraint $(X_{e_3} \vee X_{e_9} \vee X_{e_{13}})$ for the cycle between the `i=1` and `i=2` iterations by conflicts $e_3 \mapsto e_9$ and $e_9 \mapsto e_{13}$. This constraint enforces that at least one of the variables X_{e_3}, X_{e_9}, and $X_{e_{13}}$ be 1 in the solution.
2. Each event e can be made irrelevant only if all events that are data or control dependent on e are also irrelevant. For example, e_3 can be made irrelevant only if e_4, e_{13}, and e_{14} are made irrelevant, as well. For example, $(X_{e_3} \implies X_{e_4})$ is among the constraints added to model this requirement. The constraint enforces that whenever X_{e_3} is 1 in the solution, X_{e_4} be also 1 in the same solution.
3. For each event e, we add a constraint indicating that e is made irrelevant only if some **`if(*)`** is added such that both: (a) some dynamic instance of the **`if(*)`** contains e, and (b) no event contained by that dynamic instance is relevant.

   For example, an **`if(*)`** around line 5, lines 5–7, or lines 5–10 would make e_3 irrelevant, because none of events e_4, e_{13}, or e_{14} (which depend on e_3) are relevant. But an **`if(*)`** around the entire **`while`** statement would not, because the dynamic **`if(*)`** containing e_3 would also contain the relevant event e_{18}.
4. Finally, we forbid adding overlapping **`if(*)`** constructs. For example, we forbid adding both an **`if(*)`** around lines 5 and 6 and one around lines 6 and 7, as this would not be a well-structured program.

These constraints allow any solution that covers all of lines 5–10, and no more, with some number of **`if(*)`** constructs. (Because events e_3, e_4, e_{13}, and e_{14} all must be made irrelevant, and any larger **`if(*)`** including these events would include relevant events). The minimal such solution places a single **`if(*)`** that encloses lines 5-10. Thus, our algorithm produces the correct NDSeq specification for this example.

3. Background: NDSeq Specifications

In this section, we formalize the condition for a parallel program to satisfy its NDSeq specification, and formally describe our approach in [4] to checking this condition. For this, we will assume that we are given an NDSeq specification a priori. We show in the next section how to infer such a specification with minimal nondeterminism.

3.1 NDSeq Specifications and Parallelism Correctness

Given a parallel program $\mathcal{P}$, we specify the correct behavior of a parallel program by writing an equivalent nondeterministic sequential (NDSeq) program as a specification of $\mathcal{P}$, instead of explicitly giving a specification of the required input-output behavior. Informally, the equivalence means that for any input and thread schedule of $\mathcal{P}$ there exists an execution of the NDSeq program that produces the same output. We formalize these concepts next after describing the language constructs that we use to write parallel programs and their specifications.

We will follow the approach in [4] where the specification is embedded in the parallel program itself. This embedding enables one to readily apply standard conflict serializability checking techniques to verify the program against its NDSeq specification. The syntax for the language is shown in Figure 5. To simplify the presentation we consider a NDSeq program $\mathcal{P}$ to consist of a single procedure. We omit the discussion of multiple procedures and object-oriented concepts, and assume that each global variable (in $Global$) refers to a distinct location on the shared heap, and each local variable (in $Local$) refers to a distinct stack location of a thread.

$$g \in Global \quad l \in Local \quad x \in Var = Global \cup Local$$

$$\begin{aligned} s \in Stmt ::= {} & l = l \; op \; l \mid l = \text{constant} \mid l = l \mid g = l \mid l = g \mid s; s \\ & \mid \texttt{if}(l)\; s\; \texttt{else}\; s \mid \texttt{while}(l)\; s \mid \texttt{foreach}\; (l\; \texttt{in}\; l)\; s \\ & \mid \texttt{coforeach}\; (l\; \texttt{in}\; l)\; s \mid \texttt{cobegin}\; s; ...; s \\ & \mid \texttt{atomic}\; s \mid \texttt{if(*)}\; s \end{aligned}$$

Figure 5. Selected statements of our language. The constructs with a different semantics in the parallel program and the NDSeq specification are shown in gray color.

Given a parallel program $\mathcal{P}$, the NDSeq specification is obtained by (a) overloading the parallel constructs that create threads (**coforeach** and **cobegin**) in a sequential context, and (b) introducing nondeterministic control flow with **if(*)**. Specifically, given an statement s in the parallel program, the user can modify the statement to **if(*)**$\{s\}$ in the NDSeq specification. For each program $\mathcal{P}$, given a set of statements to enclose with **if(*)**, we define two sets of executions *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$), described below. The correctness of the parallel program is then given by relating *ParExecs*($\mathcal{P}$) and *NdSeqExecs*($\mathcal{P}$).

Parallel executions: *ParExecs*($\mathcal{P}$) contains the parallel executions of $\mathcal{P}$ where each **cobegin** and **coforeach** statement creates implicitly new threads to execute its body. **cobegin** $s_1; ...; s_n$ is evaluated by executing each of $s_1, ..., s_n$ on a separate, newly created thread. **coforeach** is evaluated by executing each iteration of the loop on a separate, newly created thread. Following structured fork/join parallelism, a parallel execution of a **cobegin** and **coforeach** statement terminates only after all the threads created on behalf of the statement terminate. Assignments, branch conditions, and the entire **atomic** statement are executed as atomic steps without being interrupted by other threads. Note that, **if(*)**$\{s\}$ statements are introduced in the NDSeq specification and do not appear in the parallel program.

NDSeq executions: *NdSeqExecs*($\mathcal{P}$) contains the (nondeterministic) sequential executions of $\mathcal{P}$ where all statements are evaluated by a single thread. Under the sequential semantics, the statements other than **cobegin** and **coforeach** are interpreted in the standard way. Statement **atomic** s is simply equivalent to s. Each evaluation of **cobegin** $s_1; ...; s_n$ is equivalent to running a nondeterministic permutation of statements $s_1, ..., s_n$. A statement **coforeach** is evaluated similarly to its deterministic version (**foreach**) except that the elements of the collection being iterated over are processed in a nondeterministic order. This, in essence, abstracts the semantics of the collection to an unordered set. Finally, **if(*)** indicates a nondeterministic branch. That is, each time a statement **if(*)**$\{s\}$ is evaluated, a boolean value is chosen for * nondeterministically.

The parallelism correctness for $\mathcal{P}$ means that every final state reachable by a parallel execution of the program from a given initial state is also reachable by a NDSeq execution from the same initial state. Therefore, parallel executions have no unintended nondeterminism caused by thread interleavings: either the nondeterminism is prevented using synchronization, or it is expressed by the nondeterministic control flow in the sequential specification.

While defining correctness, we distinguish a set of global variables as *focus* variables, which are considered to be effective on the functionality of the program. Then, we reason about the equivalence executions by referring to the final valuation of the focus variables. For example, consider a parallel search algorithm. The variable pointing to the best (optimal) solution found is a focus variable, while statistics counters, which do not affect the final outcome of the search, are non-focus variables.

Definition 1 (Safe parallel execution)**.** *A parallel execution in ParExecs($\mathcal{P}$) of a program $\mathcal{P}$ is* safe *with respect to a set Focus $\subseteq$ Global, if there exists a sequential execution in NdSeqExecs($\mathcal{P}$), such that the initial states are the same in both executions and their final states agree on the value of all variables $g \in Focus$.*

Definition 2 (Parallelism Correctness)**.** *A program $\mathcal{P}$* obeys *its NDSeq specification with respect to a set Focus $\subseteq$ Global, if every parallel execution of $\mathcal{P}$ is safe with respect to Focus.*

3.2 Conflict Serializability Checking of Parallel Executions

A key motivation behind keeping the specification of a parallel program similar to the parallel program is that we can readily apply standard conflict serializability checking techniques to verify the program against its NDSeq specification. We briefly describe conflict serializability checking [12, 14, 27] because we will be using such checking in our proposed inference algorithm.

We assume that the parallel program is free of low-level data races [26]. We check conflict serializability on an execution trace described as a sequence of execution *events*. For each event e we have the following information:
Type(e) is the type of the event, defined as follows:

$$T \quad ::= \quad x = x' \mid branch(l)$$

The "$x = x'$" event type corresponds to the assignment and binary operation statements in our language (shown in Figure 5; recall that metavariable x stands for both locals and globals). We use a simple assignment in our formal description to simplify the presentation; unary and binary operators do not pose notable difficulties. We assume that an event can read a global, or write a global, but not both. The "*branch*(l)" event marks the execution of a branch operation when the boolean condition denoted by local l evaluated to *true*. A branch event can be generated by the statements of the form **while**(l) s and **if**(l) s. The case of a branch when the negation of a local is *true* is similar. Our algorithm does not require specific events to mark the start and end of procedures or atomic blocks. We write $e : T$ when e has type T.
Thread(e) denotes the thread that generates the event e. Recall that a new thread is created for each dynamic instance of a block of a **cobegin** statement and for each iteration of a **coforeach** statement.

First we define the conflict relation between individual events with respect to a set of events, denoted $\mathcal{E}$. For traditional conflict serializability checking $\mathcal{E}$ is instantiated to the set of all events in a trace. However, we will overload this parameter in the subsequent sections.

Definition 3 (Conflicting events in a set of events $\mathcal{E}$)**.** *Given a set of events $\mathcal{E}$, two events $e, e' \in \tau$ are conflicting in $\mathcal{E}$ (written $e \rightsquigarrow_{\mathcal{E}} e'$) iff (a) $e, e' \in \mathcal{E}$, and (b) e occurs before e' in τ, and (c) both events operate on the same shared global variable, and at least one of them represents a write, and (d) the events are generated by different threads.*

Next we lift the conflict relation from events to threads. When comparing two threads for conflicts we need to consider their events and all the events of their descendant threads. Thus, for a thread t we define its *transaction* as the set of events *Trans* (t) that includes all the events of t and of the descendant threads of t.

Definition 4 (Conflicting threads with respect to a set of events $\mathcal{E}$)**.** *Given a set of events $\mathcal{E}$, two threads t, t' are conflicting in trace τ (written $t \rightsquigarrow_{\mathcal{E}} t'$) iff (a) their transaction sets are disjoint (i.e., one is not a descendant of the other), and (b) there exist two events $e \in Trans\ (t)$ and $e' \in Trans\ (t')$ that are conflicting ($e \rightsquigarrow_{\mathcal{E}} e'$). The relation $t \rightsquigarrow^*_{\mathcal{E}} t'$ is the transitive and reflexive closure of the thread conflict relation.*

Given a parallel execution trace τ and a subset of events $\mathcal{E}$ of τ, the runtime conflict serializability checking algorithm works as follows. We compute conflict relation $\rightsquigarrow_{\mathcal{E}}$ between threads from τ. If there exists a cycle $t \rightsquigarrow^*_{\mathcal{E}} t' \rightsquigarrow^*_{\mathcal{E}} t$ (where $t \neq t'$), then we report that the trace is not conflict-serializable; otherwise, we declare that the execution is safe. Intuitively, conflict serializability relies on the fact that an interleaved trace can be transformed incrementally into a serialized trace by a sequence of swaps of adjacent events. From the definition of a conflict, it is always safe to swap two adjacent non-conflicting events without changing the final state of the execution, and a cycle of conflicts indicates that a serialized trace cannot be obtained by swapping non-conflicting events.

The following theorem states that conflict serializability provides a sound way of checking parallel executions against a NDSeq specification. (The proof of the theorem relies on classical results from the database theory [27].)

Theorem 1. *Let τ be a parallel execution of $\mathcal{P}$. If the above runtime conflict serializability checking algorithm (where $\mathcal{E}$ contains all events in τ) does not report any non-serializable transaction, then τ is safe with respect to its NDSeq specification.*

3.3 Improving Conflict Serializability Checking

A standard conflict-serializability algorithm [27] considers all events in a trace. (Note that, Theorem 1 holds when we instantiate $\mathcal{E}$ to the set of all events of the trace.) However, in many concurrent programs it is common that results of some computations accessing shared variables are discarded and are not relevant to the rest of the execution. In Section 2 we demonstrated such a case, in which partial work based on a previous read of shared variable `x` was discarded when another conflicting access to `x` was later detected. Informally, an event is *relevant* if it writes to a memory location that is eventually used in the computation of a final value of a focus variable or a deterministic (**`while`**(l) or **`if`**(l)) branch taken in the execution. We showed that if we perform traditional serializability checking of a trace considering only the relevant events in the trace, we can safely ignore conflict cycles [4]. Below we formally define the set of relevant events; in Section 4 we show how to compute this set by constraint solving.

Let $\mathcal{S}$***** be the set of statements that are immediately enclosed with **`if(*)`** in the NDSeq specification of the program. Given a trace τ, we denote the set of relevant events in τ with respect to the NDSeq specification by $Relevant(\tau, \mathcal{S}\textbf{*})$. In order to define $Relevant$ formally, we need some notation.

Dynamic Data Dependence: To track the relevance aspect we compute a dynamic data-dependence relation between events. For trace τ, we define the data-dependence relation $\dashrightarrow_\tau$ as follows:

D1 **(Data Dependence).** For each local variable read $e_j : x = l$ or branch e_j : *branch*(l), we add a dependence $(e_i \dashrightarrow_\tau e_j)$ on the last $e_i : l = x'$ that comes before e_j in τ. This dependence represents an actual data flow through local l from e_i to e_j in the current trace. Both of these events are in the same thread (since they operate on the same local) and their order and dependence will be the same in any serialization of the trace. These dependence edges are shown as thin dashed arrows in Figure 3.

D2 **(Inter-Thread Dependence).** For each global variable read $e_j : l = g$ we add dependencies $(e_i \dashrightarrow_\tau e_j)$ on events $e_i : g = l'$ as follows. From each thread we pick the last write to g that comes before e_j in τ, and the first write to g that comes after e_j in τ. This conservative dependence is necessary because the relative order of reads and writes to a global variable from different threads may change in a serialization of the trace. In this way, dependencies are preserved while reordering the accesses.

Let $\dashrightarrow^*_\tau$ denote the transitive closure of $\dashrightarrow_\tau$. We omit the subscripts when τ is clear from context.

We write E_s to denote the set that contains exactly the events generated by a dynamic instance of s in some execution. Let $NdBlock(e)$ return the smallest set E_s such that $e \in E_s$ and $s \in \mathcal{S}$*****. In other words, $NdBlock(e)$ gives the execution of the smallest statement that generated e and is enclosed with **`if(*)`** in the NDSeq specification of the program. If e is not generated by a statement enclosed with **`if(*)`**, then $NdBlock(e)$ is undefined.

Definition 5 (Relevant events)**.** *Let τ be a parallel execution trace. $Relevant(\tau, \mathcal{S}\textbf{*})$ is the smallest set of events from τ such that $e \in Relevant(\tau, \mathcal{S}\textbf{*})$, if one of the following holds:*

R1 *$e : g = l$, and e is the last write to $g \in$ Focus in Thread(e).*

R2 *e : branch(l), and $NdBlock(e)$ is undefined.*

R3 *e : branch(l), $NdBlock(e) = E_s$, and there is an event $e' \in E_s$ such that $e' \in Relevant(\tau, \mathcal{S}\textbf{*})$.*

R4 *Exists an event $e' \in Relevant(\tau, \mathcal{S}\textbf{*})$ such that $e \dashrightarrow^*_\tau e'$. (Note that, in this case e is an assignment event.)*

R1 makes all final writes to focus variables relevant. R2 and R3 state the condition for a (deterministic) branch event e to become relevant: either if e is not part of any execution of a statement enclosed with **`if(*)`** in the NDSeq specification, or if the smallest such execution generating e already contains another relevant event. R4 makes an event relevant if it flows into another relevant event. Notice that, a dynamic instance of a statement $s \in \mathcal{S}$***** becomes totally irrelevant, if it does not contain a final write to a focus variable and none of the events generated by that instance flow (through $\dashrightarrow$) into other relevant events outside the instance.

The following theorem, proved in [4], states that it is sound to check conflict serializability by only considering relevant events in the trace. Here, we substitute $\mathcal{E}$ with $Relevant(\tau, \mathcal{S}\textbf{*})$ in Definition 3 and 4 rather than all the events in the trace.

Theorem 2 (Soundness)**.** *Let τ be the trace of a parallel execution of a program $\mathcal{P}$. If the runtime conflict serializability checking algorithm only considering $Relevant(\tau, \mathcal{S}\textbf{*})$ does not report any non-serializable transaction, then the parallel execution is* safe *with respect to the NDSeq specification of $\mathcal{P}$ and Focus.*

4. Inferring a Suitable NDSeq Specification

Having explained the runtime approach to reasoning about whether a program obeys its NDSeq specifications and the role of **`if(*)`** annotations in this, the next question is where users should add such **`if(*)`** annotations? In this paper, we are not going to describe where to insert such annotations. A list of recipes for inserting such annotations can be found our previous work [4]. In this work, we are going to figure the annotations automatically. In particular, our goal is, given a set of traces $\mathcal{T}$ of $\mathcal{P}$, to come up with a set $\mathcal{S}$***** of statements so that if we immediately enclose the statements in $\mathcal{S}$***** with **`if(*)`** and compute the relevant events in $\mathcal{T}$ (as in Definition 5), conflict serializability checking over these relevant events gives no serializability violations, showing that all the traces in $\mathcal{T}$ are safe with respect to the NDSeq specification obtained from $\mathcal{S}$*****. We assume that the user still indicates which variables constitute the focus variables.

In order to infer the NDSeq specification, we compute the set of relevant events by solving a Boolean Satisfiability problem (SAT). In contrast to the original algorithm for computing $Relevant(\tau, \mathcal{S}\textbf{*})$ in [4], formulating our reasoning as a constraint solving problem allows us to not only check that a given trace is safe. It also enables us to perform such checking without giving an NDSeq specification. Instead, we ask the SAT solver to find a suitable set $\mathcal{S}$***** of statements to enclose with **`if(*)`**, so that the solver can compute a proper set of relevant events using which it can show that the traces in $\mathcal{T}$ are all safe.

We also need to be careful with the set $\mathcal{S}$*****, because **if(*)**s could add extra behaviors to the NDSeq specification of the program and those extra behaviors should not violate the functional correctness of the program. Therefore, we need to find the minimal set $\mathcal{S}$***** that is necessary to show that the traces in $\mathcal{T}$ are all safe. For this, we then turn the problem into a Minimum-Cost Boolean Satisfiability Problem (MinCostSAT).

4.1 SAT Formulation for Inferring NDSeq Specification

We start with a program $\mathcal{P}$ where a set *Focus* of focus variables are marked by the programmer, but no statement is enclosed with **if(*)**, i.e., the set $\mathcal{S}$***** is empty. We are given a set $\mathcal{T}$ of parallel execution traces and a set of conflict cycles detected by applying a standard conflict serializability check on each trace. Theorem 1 and 2 imply that if a conflict serializability check, over all events or only the events in $Relevant(\tau, \mathcal{S}\text{*})$ where $\mathcal{S}\text{*} = \emptyset$, respectively, finds no conflict cycles in a trace τ, then τ is safe. Thus, we assume that each trace in $\mathcal{T}$ contains at least one conflict cycle. Our goal is to determine if there exist an NDSeq specification with a non-empty set $\mathcal{S}$***** of statements to enclose with **if(*)** using which we can show that all the conflict cycles can be ignored safely, i.e., each cycle contains at least one relevant event. For this, we construct and solve a SAT instance on the following (boolean) indicator variables:

- X_s, for each statement s in $\mathcal{P}$.
- X_e, for each event e in any of the traces in $\mathcal{T}$. Note that these dynamic events are uniquely identified, both in a trace and across all traces.
- X_{E_s}, for each dynamic execution of a statement s generating exactly the events in E_s in any of the traces in $\mathcal{T}$.

Let $\mathbf{X}$ denote a solution to a SAT instance. We refer to the values (from the set $\{0, 1\}$) of indicator variables X_e, X_s, and X_{E_s} in solution $\mathbf{X}$ by $\mathbf{X}_e$, $\mathbf{X}_s$, and $\mathbf{X}_{E_s}$, respectively.

We will construct our constraints to guarantee that, if our instance has a solution $\mathbf{X}$, then there exists an NDSeq specification for $\mathcal{P}$ with respect to which all traces in $\mathcal{T}$ are *safe*, and the following hold:

1. The set $\mathcal{S}\text{*} = \{s \mid \mathbf{X}_s = 1\}$ contains the statements we need to surround with **if(*)** in the inferred NDSeq specification.
2. For each event e in a trace $\tau \in \mathcal{T}$, if $\mathbf{X}_e = 1$ then event e is *irrelevant* in τ, i.e., $e \notin Relevant(\tau, \mathcal{S}\text{*})$ with respect to the inferred NDSeq specification.
3. For each X_{E_s}, if $\mathbf{X}_{E_s} = 1$, then $\mathbf{X}_s = 1$, and thus, statement s will be enclosed with an **if(*)**, and that **if(*)** around s will make events in E all irrelevant (i.e., $\mathbf{X}_e = 1$ for all $e \in E$).
4. For each conflict cycle C in some trace $\tau \in \mathcal{T}$, $\mathbf{X}_e = 1$ for at least one event e in C. This means, cycle C will not be observed when checking conflict serializability over $Relevant(\tau, \mathcal{S}\text{*})$ computed using the set $\mathcal{S}$***** given by solution $\mathbf{X}$.

We construct the full SAT instance as follows. The conditions R1 through R4 are from Definition 5 of relevant events. For simplicity, we use implications in constraints; each constraint can be trivially translated to the clause form (disjunction of literals) by using the equivalence $(X_1 \implies X_2) \equiv (\neg X_1 \vee X_2)$.

(A) Condition R1 dictates that if an event e is a final write to a variable in *Focus*, then e is relevant. Thus, for each such event e, X_e must be 0 in the solution. We ensure this by substituting 0 for each such X_e in the rest of the formulation.

(B) Conditions R2 and R3 dictate that a branch event e –i.e., of type *branch*(l) for some local l– becomes irrelevant only if (a) e is generated by a dynamic instance of a statement s which is directly enclosed by an **if(*)** and (b) all the events generated by that instance are irrelevant. To ensure (a) and (b) we add the following constraints.

- For each branch event e we add the constraint:
$$(X_e \implies \bigvee_{e \in E_s} X_{E_s}) \tag{1}$$
- For each dynamic instance of statement s, producing events E, we add the constraint:
$$(X_{E_s} \implies X_s) \tag{2}$$
- For each dynamic instance of statement s, producing events E, and for each $e \in E$, we add the constraint:
$$(X_{E_s} \implies X_e) \tag{3}$$

Therefore, if a branch event e needs to be irrelevant, the solver must find a dynamic instance of some s with all its events (including e) are irrelevant, and s must be enclosed with **if(*)**.

(C) Condition R4 dictates that if an event e' is relevant and if there is another event e such that $e \dashrightarrow^*_\tau e'$, then e must also be relevant. In other words, if e needs to be irrelevant, e' must be irrelevant, too. To ensure this, we add the following constraint for each pair of events e, e' such that $e \dashrightarrow^*_\tau e'$ for some τ:
$$(X_e \implies X_{e'}) \tag{4}$$

(D) Given two statements s and s', we say that s *overlaps* with s' if s is not nested inside s', s' is not nested inside s, and there is a statement s'' nested inside both s and s'.

If we have two overlapping statements, then we cannot surround both of them by **if(*)** because such an action would result in an invalid program. For example, consider the sequential composition of three statements $s1; s2; s3$. Then statements $s1; s2$ and $s2; s3$ overlap with each other and it is easy to see that we cannot surround both of them with **if(*)** simultaneously. Our constraint system ensures this restriction by adding the following constraint for every pair of overlapping statements s, s':
$$(X_s \implies \neg X_{s'}) \tag{5}$$

(E) Finally, we need to ensure that at least one event in each conflict cycle must be irrelevant so that we can use Theorem 2 to conclude that all the traces in $\mathcal{T}$ are safe with respect to the inferred NDSeq specification. To ensure this, for each set C of events forming a conflict cycle in a trace $\tau \in \mathcal{T}$, we add constraint:
$$\bigvee_{e \in C} X_e \tag{6}$$

If the above constraint system is satisfiable, then a solution to it gives us an NDSeq specification. In particular, if $\mathbf{X}$ is a solution to the constraint system, then $\mathcal{S}\text{*} = \{s \mid \mathbf{X}_s = 1\}$ is the set of statements to enclose with **if(*)** in the NDSeq specification. Then, all the traces in $\mathcal{T}$ are safe with respect to the inferred specification.

Theorem 3 (Inference)**.** *Given a set $\mathcal{T}$ of parallel execution traces of program $\mathcal{P}$ and focus variables Focus, assume that our SAT instance is satisfiable, and let $\mathcal{S}$***** be the set of statements, inferred from the solution, to be enclosed with* **if(*)***. Then, every trace in $\mathcal{T}$ is* safe *with respect to Focus and the inferred NDSeq specification.*

We give the proof of Theorem 3 in Appendix A.

A noteworthy implication of this theorem is that, if there is no way to show that a trace in $\mathcal{T}$ is safe by only adding **if(*)**s to statements, the solver reports that the solution is unsatisfiable.

Thus, an unsatisfiable instance indicates that one of the traces in $\mathcal{T}$ is likely to contain parallelism errors, such as atomicity violations.

4.2 MinCostSAT Solving for a Minimal NDSeq Specification

The SAT formulation above guarantees that if there is an NDSeq specification, in which a set $\mathcal{S}$* of statements are enclosed in `if(*)`, that can show that the traces in $\mathcal{T}$ are safe, then there exists a solution $\mathbf{X}$ that selects $\mathcal{S}$*, i.e., forall $s \in \mathcal{S}$*, $\mathbf{X}_s = 1$. But, we have not guaranteed that such a solution selects exactly $\mathcal{S}$*. In other words, the solution may tell us to enclose with `if(*)` more statements than those in $\mathcal{S}$*. In this case, there is a risk that some statements can be unnecessarily surrounded by `if(*)`. Since adding `if(*)` may cause adding more behaviors to the program, one could end up adding `if(*)`s that violate functional correctness. Therefore, we need a mechanism to find a solution to the constraint system so that functional correctness is not broken. We noticed that if we add a minimal number of `if(*)`s then there is a lower chance of breaking the functional correctness. For this, we re-formulate the SAT problem above as a MinCostSAT problem.

MinCostSAT is a special form of SAT, where, in addition to the constrains above, a cost function $\mathbf{C}$ assigns each variable a non-negative cost. The solver is asked to find a solution that not only satisfies all the constraints, but also minimizes the sum of the costs of the variables that are assigned 1 in the solution.

Our MinCostSAT formulation contains all the constraints (A)-(E) given above. In addition, we define the cost function $\mathbf{C}$ such that for each X_s, $\mathbf{C}(X_s) = 1$, and for other variables $X_\bullet$, $\mathbf{C}(X_\bullet) = 0$. Therefore, the solver optimizes the objective [1]

$$\textit{minimize} \sum_{s \ in \ \mathcal{P}} \mathbf{X}_s \tag{7}$$

In this formulation, X_s is assigned 1 only when a branch event e must be marked irrelevant to discharge a conflict cycle, and for this, s must be surrounded with `if(*)`. Otherwise, X_s is assigned 0 to minimize the cost.

Note that, adding only the minimum number of necessary `if(*)`s to the inferred NDSeq specification is a heuristic to reduce the risk of violating the functional specification. In other words, if we find a solution to our MinCostSAT formulation above, then we have inferred a *likely* NDSeq specification of the parallel program, and that specification may still violate the functional correctness specification of the program. If we find no solution, then probably there is no NDSeq specification for the parallel program. Thus, we foresee a repetitive process for finding the right NDSeq specification, in which the user sequentially checks the functional correctness specification (e.g., assertions) after inferring an NDSeq specification, and if any functional correctness criterion is violated, rules out the current placement of `if(*)`s for the next iteration of NDSeq specification inference.

4.3 Optimizations

We conclude this section by presenting two optimizations that we observed to have significant effect in simplifying the constraint system, and thus reducing the MinCostSAT solving time from minutes to several seconds.

Using Dynamic Slicing: Recall that in Constraint (A) we pre-assign 0 to each X_e if e is a final write to a focus variable; the values for indicator variables of other events are computed during the SAT solving. By using the dynamic slice [1] of the trace, we can improve this by providing values for more variables before the SAT solving.

Let $\hookrightarrow_\tau$ denote the control dependence between the events in trace τ, and $\longrightarrow^*_\tau$ denote the transitive closure of $(\dashrightarrow_\tau \cup \hookrightarrow_\tau)$. Recall that $\dashrightarrow_\tau$ denotes data dependence defined in Section 3.3, and see Appendix B for the formal definition of $\hookrightarrow_\tau$.

A *dynamic slice* of a trace τ with respect to the focus variables, denoted by $DSlice(\tau)$, is the set of events from τ such that $e \in DSlice(\tau)$ *iff* there exists an event $e' : g = l$ in τ, e' is the last write to $g \in$ *Focus* in *Thread*(e'), and $e \longrightarrow^*_\tau e'$.

Lemma 1. $\forall \mathcal{S}$*. $DSlice(\tau) \subseteq Relevant(\tau, \mathcal{S}$*$)$.

We give the proof of the lemma in Appendix B.

Lemma 1 indicates that given an input trace $\tau \in \mathcal{T}$ to our MinCostSAT formulation, the set of relevant events in τ given by a solution will always be a superset of the dynamic slice of τ; this result holds for any inferred NDSeq specification. In other words, if $e \in DSlice(\tau)$ then it must be relevant. Thus, we can safely modify (A), which pre-assigns 0 to X_e only if e is a final write to a focus variable, in Section 4.1 as follows.

(A') For each event $e \in DSlice(\tau)$, X_e must be 0 in the solution. Thus, we substitute 0 for each such e in the constraint system.

Grouping Events: The formulation of MinCostSAT in Section 4 considers all the events in the execution and the dependencies between those events. This could lead to large MinCostSAT instances that are expensive to solve. We address this situation by grouping events into disjoint sets. Whenever we see an execution of a statement that is completely excluded from the dynamic slice, we treat all the events $\mathbf{e} = \{e_1, ..., e_n\}$ in that dynamic execution instance of the statement as a single (compound) event. For each $i \in [1, n]$, we then replace the variable X_{e_i} in all constraints of the MinCostSAT by the variable $X_\mathbf{e}$ and we lift the constraints described in Section 4 to sets of events. In this way, we can ignore the dependency relationship between the events within a group and concentrate on inter-group dependencies. Although grouping events in this way may result in less optimal solutions (with higher cost than the original and more general formulation in Section 4), in our experiments, we confirmed that it does not affect the final solution for our benchmarks.

5. Experimental Evaluation

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 we have previously and manually written NDSeq specifications [4]. We compared the quality and accuracy of our automatically-inferred `if(*)`s to the ones in their manually-written NDSeq specifications.

Our prototype tool NDETERMIN uses bytecode instrumentation via `Soot` [32]. During the instrumentation phase, we compute the control dependencies between statements and identify the candidate static blocks that could be annotated with `if(*)`. While Section 4 describes our algorithm over structured statements, our implementation handles unstructured statements of Java. We use active random testing to generate parallel execution traces, and use the MinCostChaff [17] and MiniSat+[9] solvers to solve the MinCostSAT instances generated from these traces.

[1] This formulation can also be mapped to a Partial Maximum Satisfiability problem (PMAX-SAT), which contains our constraints in (A)-(E) as hard constraints and for each variable X_s a soft constraint $(\neg X_s)$; the objective is to satisfy all hard constraints and maximum number of soft constraints.

Benchmark		Benchmark Description	Approximate Lines of Code (App + Lib)	# of Parallel Constructs	Size of Manual NDSeq Spec		Size of Trace (Events)		Inferred NDSeq Specification	
					# of `if(*)`'s	# focus stmts	All	Sliced Out	# of `if(*)`'s	Correct?
JGF	`sor`	successive over-relaxation	300	1	0	1	905k	561k	0	yes
	`matmult`	sparse matrix-vector multiplication	700	1	0	1	962k	8k	0	yes
	`series`	coefficients of Fourier series	800	1	0	5	2008k	1215	0	yes
	`crypt`	encryption and decryption	1100	2	0	3	493k	100k	0	yes
	`moldyn`	molecular dynamics simulation	1300	4	0	1	4517k	4300k	0	yes
	`lufact`	LU factorization	1500	1	0	1	1048k	792k	0	yes
	`raytracer`	ray tracing	1900	1	0	1	9125k	8960k	-	-
	`raytracer (fixed)`	corrected ray tracing	1900	1	0	1	9125k	8960k	0	yes
	`montecarlo`	Monte Carlo derivative pricing	3600	1	0	1	1723k	731k	0	yes
PJ	`pi3`	Monte Carlo approximation of π	150 + 15,000	1	0	1	1062k	141	0	yes
	`keysearch3`	cryptographic key cracking	200 + 15,000	2	0	4	1062k	1049k	0	yes
	`mandelbrot`	fractal (Mandelbrot set) rendering	250 + 15,000	1	0	6	576k	330k	0	yes
	`phylogeny`	branch-and-bound search	4400 + 15,000	2	3	8	29k	24k	-	-
	`phylogeny (fixed)`	corrected branch-and-bound search	4400 + 15,000	2	3	8	29k	24k	1	yes
	`stack`	Treiber non-blocking stack [30]	40	1	2	8	1050	356	2	yes
	`queue`	non-blocking queue [25]	60	1	2	8	325	114	2	yes
	`meshrefine`	Delaunay mesh refinement	1000	1	2	50	930k	845k	2	yes

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

Limitations: In Java, it is necessary to handle language features such as objects, exceptions, casts, etc. While our implementation supports many intricacies of the Java language, it has a couple of limitations. First, our implementation tracks neither the shared reads and writes made by uninstrumented native code, nor the flow of data dependence through such native code. Second, in order to reduce the runtime overhead, our tool does not instrument all of the Java standard libraries. Thus, we could miss conflicts or data dependencies carried out through the native code and the Java libraries, and fail to include some events in our inference algorithm. To address the second limitation, for certain shared data structure objects we introduced shared variables and inserted reads or writes of those variables whenever their corresponding objects were accessed. This allowed us to conservatively approximate the conflicts and data dependencies for certain critical standard Java data structures. We did not observe any inaccuracy in our experimental results due to these limitations.

5.1 Experimental Setup

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 [29] and the Parallel Java (PJ) Library [20]. For our benchmarks, we use the same focus variable and parallel region annotations as in [4]. (Although these benchmarks are written in a structured parallel style, they use explicit Java threads as Java does not provide `cobegin` or `coforeach` constructs. Thus, the code was annotated to indicate which regions of code correspond to the bodies of structured `cobegin`'s or `coforeach`'s.)

Note that benchmarks `raytracer` and `phylogeny` both contain parallelism errors. Thus, we apply NDETERMIN to both the original version of each benchmark, and a version in which the error has been fixed. (For `raytracer`, we modify a `synchronized` block to use the correct shared lock to protect the key global variable `checksum1`. For `phylogeny`, we make one method `synchronized` in order to eliminate an atomicity error.)

We execute each benchmark five times on a single test input, using a simple implementation of race-directed active random testing [28]. For each benchmark, NDETERMIN analyzes all five executions and either infers a placement of **`if(*)`** for the benchmark's NDSeq specification or reports that the benchmark satisfies no possible NDSeq specification due to a parallel error.

We performed our experiments on a 64-bit Linux machine with a dual Quad-Core/HT Intel(R) Xeon(R) CPU (2.67GHz) processor, 24MB L3 cache and 48GB of DDR3/1066 RAM. For each experiment, we measured the time for solving both SAT instances (without minimizing the number of **`if(*)`**s) using ZChaff and MiniSat, and MinCostSAT instances using MinCostChaff and MiniSat+ generated during the experiment. We observed that for the benchmarks that do not require nondeterministic branches in their NDSeq specifications, the solving time for SAT and MinCostSAT are very close to each other, as the solver can satisfy all the constraints without needing to optimize the number of **`if(*)`**s. For most of the benchmarks, the solving time was in terms of milliseconds, and few large benchmarks required several seconds to solve the constraints. Therefore, in the following we report on the quality and accuracy of the inferred specifications.

5.2 Experimental Results

The results of our experimental evaluation are summarized in Table 1. 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 [4], `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 our manual specifications from [4]. 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.

Discussion: 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.

6. Related Work

Several parallel correctness criteria, including data-race freedom [26], atomicity [16], linearizability [19], and determinism [2, 5] have been studied for shared memory parallel programs that separate the reasoning about functionality and parallelism at different granularities of execution. All these criteria provide the separation between parallel and functional correctness partially, as the restriction on thread interleavings is limited, for example, to atomic block boundaries. NDSeq specifications [4] develop this idea up to a complete separation between parallelism and functionality so that the programmer can reason about the intended functionality by examining a sequential, but nondeterministic, program.

Reasoning about conflicting accesses that are simultaneously enabled but ineffective on the rest of the execution is the main challenge in both static [7, 16, 31] and dynamic [3, 14, 21, 23, 33, 34] techniques for checking atomicity and linearizability. The Purity work [13] and QED [10] provide static analyses to rule out spurious warnings due to such conflicts by abstracting these operations to no-op's. Their abstraction techniques resemble identifying irrelevant events by a dependency analysis. However, lack of dynamic information during the static verification is a bottleneck in automating their overall approach

DETERMIN [6] infers likely semantic determinism specifications [5] for parallel programs. This is done by monitoring and analyzing program states during parallel executions of the program and generating pre- and post- bridge predicates [5], conjunctions of equality predicates over program variables from pairs of program executions.

There is a rich literature on generating specifications/invariants for sequential programs. Daikon [11] automatically infers likely program invariants using statistical inference from a program's execution traces. Csallner et al. [8] propose an approach, called DySy, that combines symbolic execution with dynamic testing to infer preconditions and postconditions for program methods. Hangal and Lam [18] propose DIDUCE, which uses online analysis to discover simple invariants over the values of program variables. Deryaft [24] is a tool that specializes in generating constraints of complex data structures. Logozzo [22] proposed a static approach that derives invariants for a class as a solution of a set of equations derived from the program source. Houdini [15] is an annotation assistant for ESC/Java. It generates a large number of candidate invariants and repeatedly invokes the ESC/Java checker to remove unprovable annotations, until no more annotations are refuted.

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 CNS-0720906, 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] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation (PLDI)*, 1990.

[2] 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 *Object-Oriented Programming, Systems, Languages, and Applications OOPSLA*, pages 97–116, 2009.

[3] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *Programming Language Design and Implementation (PLDI)*, 2010.

[4] 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.

[5] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *Foundations of Software Engineering (FSE)*, 2009.

[6] J. Burnim and K. Sen. DETERMIN: inferring likely deterministic specifications of multithreaded programs. In *32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[7] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer Aided Verification (CAV)*, 2006.

[8] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *30th ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[9] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[10] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Principles of Programming Languages (POPL)*, pages 2–15, 2009.

[11] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, June 2000.

[12] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, pages 52–65, 2008.

[13] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[14] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Programming language design and implementation (PLDI)*, 2008.

[15] C. Flanagan and R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe (FME)*, 2001.

[16] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation (PLDI)*, 2003.

[17] Z. Fu and S. Malik. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, 2002.

[19] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12:463–492, July 1990.

[20] 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.

[21] Z. Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering*, pages 235–244, 2010.

[22] F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, January 2004.

[23] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[24] M. Z. Malik, A. Pervaiz, , and S. Khurshid. Generating representation invariants of structurally complex data. In *TACAS*, pages 34–49, 2007.

[25] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Principles of Distributed Computing (PDOC)*, 1996.

[26] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Prog. Lang. Syst.*, 1(1):74–88, 1992.

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

[28] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

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

[30] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[31] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Verification, Model Checking, and Abstract Interpretation*, 2009.

[32] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON*, 1999.

[33] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32:93–110, 2006.

[34] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993.

A. Correctness of Inference Algorithm

Let $\mathcal{T}$ be a set of parallel execution traces of program $\mathcal{P}$, and *Focus* be a set of focus variables. Let $\mathbf{X}$ be a satisfying solution to the SAT formulation in Section 4.1, and $\mathcal{S}$***** contain the set of statements to enclose with **`if(*)`** in the inferred NDSeq specification, i.e., $s \in \mathcal{S}$***** *iff* $\mathbf{X}_s = 1$.

Note that, we have already proved (in [4]) Theorem 2, which says that checking conflict serializability of each trace τ in $\mathcal{T}$ by only considering $Relevant(\tau, \mathcal{S}$*****$)$ is sound: if we find no conflict cycles after omitting irrelevant events, then τ is safe with respect to the inferred NDSeq specification. Therefore, in the following it is enough to show that the solution to the SAT formulation gives us a superset of $Relevant(\tau, \mathcal{S}$*****$)$, that is, if $e \in Relevant(\tau, \mathcal{S}$*****$)$, then the solution indicates that e is relevant.

Lemma 2. *Let $\mathcal{T}$ be a set of parallel execution traces of program $\mathcal{P}$. Let $\mathcal{S}$***** be a set of* **`if(*)`** *inferred by the above algorithm, given $\mathcal{T}$ and focus variables Focus. Let $\mathcal{S}$***** correspond to solution $\mathbf{X}$ of the constraint system built by the inference algorithm.*

*For each event e in $\tau \in \mathcal{T}$, if $e \in Relevant(\tau, \mathcal{S}$*****$)$, then $\mathbf{X}_e = 0$.*

Proof. Recall the conditions R1-R4 in Definition 5 for an event $e \in \tau$ to be in the set $Relevant(\tau, \mathcal{S}$*****$)$. Given these conditions, think of an iterative procedure to compute the relevant events: $Relevant(\tau, \mathcal{S}$*****$)$ is initialized to an empty set, and at each step, one of the rules R1-R4 is applied to add a new event to $Relevant(\tau, \mathcal{S}$*****$)$, until $Relevant(\tau, \mathcal{S}$*****$)$ does not change. We do the proof by induction on the length of this iteration. The base case where $Relevant(\tau, \mathcal{S}$*****$) = \emptyset$ is trivial. In the following, we do a case split on the condition that causes event e to be added to set $Relevant(\tau, \mathcal{S}$*****$)$.

R1 In this case, e is the last write to a focus variable by one of the threads. Therefore, by Constraint (A), $\mathbf{X}_e$ is always substituted by 0.

R2 In this case, e is a *branch*(l) event and $NdBlock(e)$ is undefined, i.e., e is not generated by a statement enclosed with **`if(*)`**. Consider each X_{E_s} term in Constraint (B.1) for e:

$$(X_e \implies \bigvee_{e \in E_s} X_{E_s})$$

For each dynamic execution E_s such that $e \in E_s$, s is not enclosed with **`if(*)`**; otherwise $NdBlock(e)$ would be defined. Thus, it must be the case that $s \notin \mathcal{S}$***** and $\mathbf{X}_s = 0$. Therefore, by Constraint (B.3), each such $\mathbf{X}_{E_s}$ is 0. Finally, by Constraint (B.1), we have $\mathbf{X}_e = 0$.

R3 In this case, e is a *branch*(l) event, $NdBlock(e) = E_s$, and there is an event $e' \in E_s$ such that $e' \in Relevant(\tau, \mathcal{S}$*****$)$. By inductive hypothesis, $\mathbf{X}_{e'} = 0$. To reach a contradiction, assume that $\mathbf{X}_e = 1$, and consider each $X_{E'_{s'}}$ term in Constraint (B.1) for e:

$$(X_e \implies \bigvee_{e \in E'_{s'}} X_{s', E'})$$

To satisfy this constraint, there must be at least one dynamic execution $E'_{s'}$ such that $\mathbf{X}_{E'_{s'}} = 1$. Moreover, by Constraint (B.3), for every $e'' \in E'_{s'}$, $\mathbf{X}_{e''} = 1$ must hold. Thus, $E'_{s'}$ cannot contain e' for which $\mathbf{X}_{e'} = 0$.
Since $\mathbf{X}_{E'_{s'}} = 1$, Constraint (B.2) ensures that $\mathbf{X}_{s'} = 1$, so s' is enclosed with **`if(*)`**. Note that, by definition of $NdBlock(e)$, s is also enclosed with **`if(*)`**. By Constraint (D.5), statements s and s' cannot overlap; either E_s and $E'_{s'}$ are disjoint or one contains the other. And E_s and $E'_{s'}$ are not disjoint, because e is in both. Again by definition of $NdBlock(e)$, E_s is the smallest nondeterministic branch containing e, so it is the case that $E_s \subseteq E'_{s'}$. Since $e' \in E$, $e' \in E'$ must hold. This contradicts with our assumptions that E' cannot contain e'.

R4 In this case, $e \dashrightarrow^* e'$ for some e' already in *Relevant*. Therefore, by inductive hypothesis, $\mathbf{X}_{e'} = 0$. Constraint (C.1) ensures that $\mathbf{X}_e = 0$.

□

Theorem 3 (Inference)**.** *Given a set $\mathcal{T}$ of parallel execution traces of program $\mathcal{P}$ and focus variables Focus, assume that our SAT instance is satisfiable, and let $\mathcal{S}$* be the set of statements, inferred from the solution, to be enclosed with* **if(*)***. Then, every trace in $\mathcal{T}$ is* safe *with respect to Focus and the inferred NDSeq specification.*

Proof. Suppose some $\tau \in \mathcal{T}$ is not safe with respect to the inferred NDSeq specification. That is, there exist events $C = e_1, \ldots, e_k$ that are all relevant and that form a cycle of conflicts between the threads of τ. Note that the $e_1, \ldots, e_k$ are all events of type "$x = x'$". (A *branch*(l) event can not appear in a cycle as l is a local variable.)

The inferred **if(*)** locations $\mathcal{S}$* correspond to a solution $\mathbf{X}$ to the constraint system built by our inference algorithm. By Lemma 2, because the $e_1, \ldots, e_k$ are all relevant, we have:

$$\mathbf{X}_{e_1} = \mathbf{X}_{e_2} = \cdots = \mathbf{X}_{e_k} \; = \; 0$$

But this contradicts Constraint (E)—that, because $e_1, \ldots, e_k$ form a conflict cycle, the solution $\mathbf{X}$ must satisfy the following constraint:

$$(\mathbf{X}_{e_1} \vee \mathbf{X}_{e_2} \vee \cdots \vee \mathbf{X}_{e_k})$$

Therefore, for each conflict cycle, at least one event in the cycle must be marked irrelevant in the solution $\mathbf{X}$. Theorem 2 states that in this case all the traces are safe because there exists no conflict cycles with all relevant events. □

B. Using Dynamic Slicing

To introduce the dynamic slicing, we need to define the control dependence relation $\hookrightarrow_\tau$ between events of a trace τ.

C1 **(Control Dependence).** For a branch evente_i : *branch*(l) and any event e_j, we add $e_i \hookrightarrow_\tau e_j$, if e_j is control dependent on e_i. Note that e_i is control dependent on e_j if and only if

1. e_i is the *branch*(l) event generated from the execution of a **if**(l) s or **while**(l) s,
2. e_j is generated by the execution of a statement s' contained in the nested s statement, and
3. no other conditional or loop in s contains the statement s'.

Let $\longrightarrow^*_\tau$ denote the transitive closure of $(\dashrightarrow_\tau \cup \hookrightarrow_\tau)$.

Dynamic Slice A *dynamic slice* of a trace with respect to the focus variables is computed as follows. We first compute the set $Target(\tau) = \{e : g = l \in \tau \mid g \in Focus \wedge$ *e* is last write to g in *Thread* (e)$\}$, i.e. the set of all events that directly affect the final output. Then a dynamic slice of a trace τ, denoted by $DSlice(\tau)$, is the set $\{e \in \tau \mid \exists e' \in Target(\tau)$ such that $e \longrightarrow^*_\tau e'\}$, i.e. the set of all events that directly or indirectly affect the final output.

The following observation enables us to perform the optimization given in Section 4.3 to improve the efficiency of solving our MinCostSAT instances.

Lemma 1. *Given an input trace $\tau \in \mathcal{T}$ to our SAT formulation, a dynamic slice of τ is always a subset of the set of relevant events given by a solution (independent of the inferred NDSeq specification). That is, $\forall \mathcal{S}$*. $DSlice(\tau) \subseteq Relevant(\tau, \mathcal{S}$*$)$.*

Proof. Note that both $DSlice(\tau)$ and our SAT instance in Section 4.1 use the same set *Focus* of focus variables given by the user. Let $\mathbf{X}$ be a solution to our SAT formulation. We show that if $e \in DSlice(\tau)$, then e cannot be marked irrelevant, i.e., $\mathbf{X}_e = 0$.

$e \in DSlice(\tau)$ holds if either e is the last write to some $g \in Focus$ or $e \longrightarrow^* e'$ for some $e' \in DSlice(\tau)$. In the former case ($e \in DSlice(\tau)$), Part (A) of the SAT formulation ensures that $\mathbf{X}_e = 0$. For the latter case ($e \longrightarrow^* e'$ and $e' \in DSlice(\tau)$) we do a proof by induction on the length of $\longrightarrow$. For the induction, we assume that e' is marked relevant, so $\mathbf{X}_{e'} = 0$. In the base case $e = e'$, and thus, $\mathbf{X}_e = 0$. In the inductive case, we rely on the contrapositive form of Lemma 3 (proved below): if e' is marked relevant, either e is marked relevant or $e \longrightarrow^* e'$ does not hold. □

Lemma 3. *Given an input trace $\tau \in \mathcal{T}$ to our SAT formulation and a NDSeq specification, if there exists two events e and e' in τ such that e is irrelevant and $e \longrightarrow^* e'$, then e' must be irrelevant.*

Proof. We prove the contrapositive form of the statement. Assume that e' is marked relevant, i.e., $\mathbf{X}_{e'} = 0$, and $e \longrightarrow^* e'$. We prove that e is marked relevant, i.e., $\mathbf{X}_e = 0$, in the solution.

We do induction on the length of $\longrightarrow^*$. The base case when $e = e'$ is trivial. Now assume that $e \longrightarrow e'' \longrightarrow^* e'$ for some e''. By inductive hypothesis e'' is also marked relevant, so $\mathbf{X}_{e''} = 0$. By definition of $\longrightarrow$, either $e \dashrightarrow e''$ or $e \hookrightarrow e''$. In the former case, Constraint (4) ensures that $\mathbf{X}_e = 0$ (thus the relations in $DSlice(\tau)$ due to D1 and D2 are respected by the solution).

Now assume the latter case: $e \hookrightarrow e''$. In this case $e =$ *branch*(l) for some l. To reach a contradiction, let us assume $\mathbf{X}_e = 1$. In this case, Constraint (1) ensure that there is some $\mathbf{X}_{E_s} = 1$ for some statement s enclosed with **if(*)**, thus $\mathbf{X}_s = 1$ also holds. In addition, by Constraint (3), $\mathbf{X}_{e'''} = 1$ must hold for all $e''' \in E$. By definition of $\hookrightarrow$ and that **if(*)**s are structured, e'' must be in E, and thus $\mathbf{X}_{e''} = 1$, contradicting with our assumption that $\mathbf{X}_{e''} = 0$. Therefore, for each solution $\mathbf{X}$ to SAT, if $\mathbf{X}_{e'} = 0$ and $e \longrightarrow^* e'$, then $\mathbf{X}_e = 0$ holds. □

C. Incorporating Functional Correctness

Note that, although the formulation above aims to add only the minimum number of necessary **if(*)**s to the inferred NDSeq specification, the risk of violating the functional specification remains. We noticed that if a minimal solution, say $\mathcal{S}$*, i.e., a solution of the constraint system that optimizes objective (7), violates functional correctness, then we can add the following constraint to the constraint system to avoid the solution:

$$\bigvee_{s \in \mathcal{S}\text{*}} \neg X_s \tag{8}$$

and solve the resultant constraint system again while optimizing the objective (7). Notice that, Constraint (8) plays the same role as a "conflict clause" in the SAT terminology and prevents the solver to assign 1 to all variables X_s for $s \in \mathcal{S}$* within the same solution. We can repeat this process until we find a solution that satisfies both functional correctness and passes serializability check or we find no solution or we run out of time. If we find a solution in this iterative process, then we have inferred a likely NDSeq specification of the parallel program. If we find no solution, then probably there is no NDSeq specification for the parallel program.