# Delay-Bounded Scheduling

A Canonical Characterization of Scheduler Nondeterminism

Michael Emmi
LIAFA, Université Paris Diderot, France
mje@liafa.jussieu.fr

Shaz Qadeer
Microsoft Research, Redmond, WA, USA
qadeer@microsoft.com

Zvonimir Rakamarić
University of British Columbia, Canada
zrakamar@cs.ubc.ca

Technical Report
MSR-TR-2010-123

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

# Abstract

We provide a new characterization of scheduling nondeterminism by allowing deterministic schedulers to delay their next-scheduled task. In limiting the delays an otherwise-deterministic scheduler is allowed, we discover concurrency bugs efficiently—by exploring few schedules—and robustly—i.e., independent of the number of tasks, context switches, or buffered events. Our characterization elegantly applies to any systematic exploration (e.g., testing, model checking) of concurrent programs with dynamic task-creation. Additionally, we show that certain delaying schedulers admit efficient reductions from concurrent to sequential program analysis.

# 1 Introduction

A concurrent program is a transition system combined with a (nondeterministic) scheduler. The program's semantics is easy to describe: the scheduler repeatedly chooses an enabled state-altering transition to execute, executes it, then chooses another transition, and so on. This inherently nondeterministic semantics is the root cause of Heisenbugs—i.e., programming errors that manifest rarely, and are hard to reproduce and repair. A class of techniques known as *model checking* [8] systematically explore this nondeterminism in order to discover, or occasionally prove the absence of, such bugs.

Although systematically exploring (or *searching*) a concurrent program's behavior is a simple and intuitively appealing idea, the exploration is computationally expensive. For programs in which the only source of nondeterminism is the scheduler, the combinatorial cost is determined by two factors: the maximum number $I$ of scheduler invocations, and the maximum number $C$ of choices available to the scheduler at each invocation. Given these two factors, the cost of exploration is $\mathcal{O}(C^I)$. While $I$ naturally corresponds to the length of program executions, $C$ corresponds to the number of concurrently executing tasks. Both $C$ and $I$ grow rapidly in realistic programs, causing a combinatorial explosion in the exploration cost.

Search prioritization is a basic strategy to combat the explosion. In general, one characterizes a subset of the search space by a bounding parameter $p$. More behaviors are explored as $p$ is increased, and in the limit all behaviors are explored. A prioritization is effective when useful information (e.g., the presence of bugs) is obtained by examining few behaviors (i.e., low values of $p$).

In this work we introduce *delay-bounding*: an effective search prioritization strategy for concurrent programs that handles both statically-known and dynamically-created tasks. We begin by quantifying scheduler nondeterminism with the concept of *delay*. Our main insight is captured by the premise:

> A <u>deterministic scheduler</u> is made sufficiently nondeterministic with the ability to delay its next-scheduled task.

In other words, we can quantify—and thus limit—scheduling nondeterminism by allotting a finite delay-budget.

More concretely, delay-bounding parameterizes the search space by a deterministic scheduler $M$, and a delay-bound $K$. When $K = 0$, exploration is limited to the unique execution produced by $M$. When $K > 0$, the scheduler may deviate from its usual schedule a total of $K$ times over an entire execution. For instance, a round-based delaying scheduler executes all of its scheduled tasks to completion in a given round $i$, before advancing to the next round, where the tasks delayed in round $i$ are again scheduled; as the total number of exercised delays is bounded by $K$, the number of rounds is bounded by $K{+}1$. The search space for any deterministic scheduler and delay-bound $K$ is bounded by $I^K$.

Delay-bounding is endowed with many appealing properties:

- Delay-bounding is a canonical characterization, and a means of limiting, scheduling nondeterminism. Since the bound is chosen independently of the number of tasks, the approach naturally handles both statically-known and dynamically-created tasks.

- The cost of delay-bounded exploration is polynomial in $I$. Our preliminary experiments discover (previously unknown) bugs in real programs with small delay-bounds, suggesting that delay-bounded search does provide adequate coverage in practice.

- The choice of a deterministic scheduler is independent of the delay-bound, and *every* reasonable scheduler discovers *any* given bug at some cost. To minimize exploration cost it is even possible to perform parallel exploration using various deterministic schedulers—perhaps even chosen at random.

- It is possible, with certain deterministic schedulers, to capture a concurrent program's delay-bounded semantics as a sequential program. Theoretically, the *scheduling complexity* of these schedulers is reduced from undecidable (with preemptible tasks), or EXPSPACE-complete (with non-preemptible tasks), to NP-complete (with or without preemption). In practice, the reduction allows us to leverage the numerous existing tools and techniques for sequential-program verification, including symbolic model checking, and symbolic exploration with SMT solvers.

## 1.1 Contributions

Our contributions are summarized as follows:

- We introduce a canonical characterization of scheduling nondeterminism, allowing a simple, elegant, and unified approach for prioritized exploration of concurrent programs with dynamic task-creation.

- We show that delaying deterministic schedulers can efficiently discover both known and previously-undiscovered bugs by systematic exploration (e.g., testing).
- We give concurrent-to-sequential program translations using clever encodings of certain delaying schedulers. The encodings rely on insightful extensions of Lal and Reps [26]'s guess and constrain methodology, and ultimately lead to practical verification algorithms.
- We identify an NP-complete analysis problem: state-reachability under a particular delaying scheduler. The complexity is lower than related bounded reachability problems.

### 1.2 Comparison to Related Search Prioritizations

Delay-bounding represents a convergence of a progression over the last decade of search prioritization techniques for concurrent programs. Given a number $I$ of scheduler invocations and $C$ of choices available to the scheduler at each invocation—recall the cost of complete search is $\mathcal{O}(C^I)$:

**Depth-bounding** limits the number of scheduler invocations by a *depth-bound* $d$, reducing the search complexity to $\mathcal{O}(C^d)$. Though this approach is taken by VeriSoft [16], it is clearly not effective as $d$ approaches $I$. In practical terms, errors manifested deeply in a program execution remain difficult to discover.

**Context-bounding [33]** labels the choices available to the scheduler by distinct (task) identifiers, and bounds the number of label changes in executions by a *context-bound* $c$. Here the complexity is reduced to $\mathcal{O}((IC)^c)$; the polynomial dependence on execution length alleviates, to a large extent, the aforementioned problem with depth-bounding. Empirically, low context-bound values are sufficient for finding many bugs in real programs [34, 26]. However, in many cases, bugs are exhibited only after each of the, say $C$, tasks have a chance to execute. (In practice this dependence arises in, e.g., initialization patterns, round-robin protocols, etc.) Consequently, the exponential dependence on $c$ in $\mathcal{O}((IC)^c)$ is practically an exponential dependence on $C$ as well. Worse yet, when tasks are created dynamically, context-bounding suffers the same ill fate as depth-bounding: errors manifested deeply in a task-creation chain are discovered only for high values of $c$.

**Preemption-bounding [28]** also labels scheduler choices with distinct identifiers, but only bounds the number of *preemptive* label changes (by $p$); label changes due to blocking or completion are not budgeted. Preemption-bounded search addresses the main problem with context-bounding, as each task gets the opportunity to complete its execution. Unfortunately, the number of label changes is *expected to* depend on $C$, rendering the search complexity, $\mathcal{O}((IC)^{p+C})$, prohibitive for large values of $C$.

As we can see, when tasks are numerous or dynamically created, the existing prioritized exploration techniques are ineffective. Delay-bounding fixes these problems by allowing an unbounded number of tasks to execute, and bounding instead the degree of variation from a deterministic scheduling order. The exploration cost using $K$ delays is $I^K$, which is independent of the number of (dynamically-created) tasks.

## 2 Asynchronous Programs

So that our approach handles dynamic task-creation *by design*, we begin with a model of simple asynchronous programs corresponding to the style of single-threaded event-driven programming. The style is typically used as a lightweight technique for adding reactivity to single-threaded applications by breaking up long-running computations into a collection of *tasks*. In this model, control begins with a non-empty *task buffer* of pending tasks from which a *dispatcher* picks a single task to execute. The dispatcher transfers control to the task, which is essentially a sequential program that can read from and write to global storage, and *post* additional tasks to the task buffer. When a task completes its execution, control returns to the dispatcher, which picks another task from the task buffer, and so on. When the dispatcher has control and the task buffer is empty, the program terminates. This model forms the basis of (client-side) web applications [14], and has been shown useful for building fast servers [31], routers [21], and embedded sensor networks [17].

In what follows we generalize the usual notion [37] of task-buffer dispatch (i.e., choosing any pending task) by exposing a *scheduler*, by which we parameterize a simple program semantics.

To capture programs with interruptible tasks (or, alternatively, concurrently running tasks on sequentially consistent processors/threads), we extend the simple asynchronous model to permit arbitrary task interruption with the **yield**-statement. Although the extension may seem modest, the resulting language is powerful enough to model concurrent programs with arbitrary preemption and synchronization, e.g., by inserting a **yield**-statement before every shared variable access. The model particularly resembles typical operating-systems kernel code and device drivers, which are often implemented as a collection of preemptible task-handlers [30, 9].

### 2.1 The Scheduler

For the moment, let $\mathsf{Tasks}$ and $\mathsf{Vals}$ be uninterpreted sets, and $\mathsf{blocked} : \mathsf{Tasks} \times \mathsf{Vals} \rightarrow \mathbb{B}$ be an uninterpreted predicate. Intuitively, each element $w \in \mathsf{Tasks}$ is a task to be scheduled, each element $g \in \mathsf{Vals}$ is a global state, and $\mathsf{blocked}(w, g)$ holds when $w$ is not enabled in the global state $g$.

A *scheduler* $M = \langle D, \mathsf{empty}, \mathsf{give}, \mathsf{take}\rangle$ consists of a datatype $D$ of *scheduler objects* $m \in D$, a scheduler *constructor* $\mathsf{empty} \in D$, and scheduler update functions $\mathsf{give} : D \times \mathsf{Tasks} \to D$ to receive posted tasks, and $\mathsf{take} : D \times \mathsf{Vals} \to \wp(D \times \mathsf{Tasks})$ to determine which tasks can be scheduled next. The scheduler $M$ is *deterministic* when for all $m \in D$ and $g \in \mathsf{Vals}$, $\mathsf{take}(m, g)$ contains at most one element, and is *non-blocking* when for all $w \in \mathsf{Tasks}$, $g \in \mathsf{Vals}$, and $m, m' \in D$, $\mathsf{blocked}(w, g)$ implies $\langle m', w\rangle \notin \mathsf{take}(m, g)$. In general, non-blocking schedulers must be aware of the program state; hence the argument $g \in \mathsf{Vals}$ to $\mathsf{take}$.

Fix $m \in D$ and $g \in \mathsf{Vals}$. A task $w$ is *pending (in $m$)* if there is a sequence $m_1 w_2 m_2 w_3 m_3 \dots w_j m_j$ such that $m_1 = m$, $w_j = w$, and for $1 < i \leq j$, $\langle m_i, w_i\rangle \in \mathsf{take}(m_{i-1}, g)$; in other words, $w$ is pending when $w$ can eventually be taken from $m$. A task $w$ is *schedulable (by $m$)* when there exists $m'$ such that $\langle m', w\rangle \in \mathsf{take}(m, g)$.

**Scheduler 1** (The bag scheduler)**.** The multiset-based scheduler $\mathsf{bag}$ is defined on the multiset domain $D_{\mathsf{bag}}$ of tasks as[1]

$$\begin{aligned} \mathsf{empty}_{\mathsf{bag}} &\overset{\text{def}}{=} \emptyset \\ \mathsf{give}_{\mathsf{bag}}(m, w) &\overset{\text{def}}{=} m \cup \{w\} \\ \mathsf{take}_{\mathsf{bag}}(m, g) &\overset{\text{def}}{=} \{\langle m \setminus \{w\}, w\rangle : w \in m\}. \end{aligned}$$

The bag scheduler is non-deterministic, since all pending tasks are returned by each application of $\mathsf{take}_{\mathsf{bag}}$.

We consider only schedulers $M$ that refine the bag scheduler, i.e., schedulers satisfying the property:

**Property 1.** *Any sequence of* $\mathsf{give}$ *and* $\mathsf{take}$ *operations allowed by $M$ are also allowed by* $\mathsf{bag}$.

Note that a scheduler can be *lossy*—i.e., can drop pending tasks—though is not allowed to "pull tasks out of thin air," i.e., schedule tasks that have not been posted. The following is an example of a lossy scheduler that refines the $\mathsf{bag}$ scheduler.

**Scheduler 2** (The bounded bag scheduler)**.** The *$K$-bounded bag scheduler* $\mathsf{bb}$ is defined on the multiset domain $D_{\mathsf{bb}}$ of tasks as

$$\begin{aligned} \mathsf{empty}_{\mathsf{bb}} &\overset{\text{def}}{=} \emptyset \\ \mathsf{give}_{\mathsf{bb}}(m, w) &\overset{\text{def}}{=} \begin{cases} m \cup \{w\} & \text{if } m(w) < K \\ m & \text{otherwise} \end{cases} \\ \mathsf{take}_{\mathsf{bb}}(m, g) &\overset{\text{def}}{=} \{\langle m \setminus \{w\}, w\rangle : w \in m\}. \end{aligned}$$

The bounded bag scheduler is non-deterministic, since all pending tasks are returned by each application of $\mathsf{take}_{\mathsf{bb}}$. It is lossy since it will drop posted tasks if its bound $K$ has been reached.

$$\begin{array}{rl} P ::= & \textbf{var } \mathtt{g} : T \ H^* \\ H ::= & \textbf{proc } \ p \ (\textbf{var } \mathtt{l} : T) \ s \\ s ::= & s; \ s \\ \mid & x := e \\ \mid & \textbf{skip} \\ \mid & \textbf{assume } e \\ \mid & \textbf{if } e \textbf{ then } s \textbf{ else } s \\ \mid & \textbf{while } e \textbf{ do } s \\ \mid & \textbf{call } x := p \ e \\ \mid & \textbf{return } e \\ \mid & \textbf{post } p \ e \\ x ::= & \mathtt{g} \mid \mathtt{l} \end{array}$$

Figure 1: The grammar of simple asynchronous programs.

## 2.2 Program Syntax

Let $\mathsf{Procs}$ be a set of procedure names, $\mathsf{Vals}$ a set of values containing `true` and `false`, and $T$ the type of values. The grammar of Figure 1 describes our language of *simple asynchronous programs*, where $p$ ranges over procedure names. We intentionally leave the syntax of expressions $e$ unspecified, though we do insist the set of expressions $\mathsf{Exprs}$ contains $\mathsf{Vals}$ and the *(nullary) choice operator* $\star$. A *simple (synchronous) program*, or *sequential program*, is a simple asynchronous program which does not contain **post** -statements.

Each program declares a single type-$T$ global variable $\mathtt{g}$, and a sequence of procedures $p_1 \dots p_n \in \mathsf{Procs}^*$. Each procedure $p$ has single type-$T$ parameter $\mathtt{l}$, and a top-level statement denoted $s_p$. The set of program statements $s$ is denoted $\mathsf{Stmts}$.

We assign the statements of simple asynchronous programs their usual meaning. In particular, **post** $p$ $e$ is an asynchronous call to procedure $p$ with argument $e$, which returns immediately with the expectation that $p$ is invoked at a later time; the **assume** $e$ statement proceeds only when $e$ evaluates to `true`. (Later on we use the **assume**-statement to block undesired executions in a reduction to sequential programs.)

## 2.3 Program Semantics

For the remainder of this section we fix a predicate $\mathsf{blocked} : \mathsf{Tasks} \times \mathsf{Vals} \to \mathbb{B}$ and a scheduler $M = \langle D, \mathsf{empty}, \mathsf{give}, \mathsf{take}\rangle$.

A *frame* $\langle \ell, s\rangle$ is a valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable $\mathtt{l}$ with a statement $s$. A *task* $w$ is a sequence of frames, and the set $(\mathsf{Vals} \times \mathsf{Stmts})^*$ of tasks is denoted $\mathsf{Tasks}$. A *configuration* $c = \langle g, w, m\rangle$ is a valuation $g \in \mathsf{Vals}$ of the global variable $\mathtt{g}$ with a task $w \in \mathsf{Tasks}$, and a scheduler object $m \in D$. We say a task $w$ is *blocked* in a

[1] Here $\cup$ and $\setminus$ are the *multiset* union and difference operators.

configuration $\langle g, w', m\rangle$ (alternatively, in a global state $g$) when $\mathsf{blocked}(w, g) = \mathtt{true}$.

For expressions without program variables, we assume the existence of an evaluation function $[\![\cdot]\!] : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ such that $[\![\star]\!] = \mathsf{Vals}$. For convenience, we define

$$e(\langle g, \langle \ell, s\rangle\, w, m\rangle) \overset{\text{def}}{=} e(g, \ell) \overset{\text{def}}{=} [\![e[g/\mathtt{g}, \ell/\mathtt{l}]]\!]$$

—since g and l are the only variables, the expression $e[g/\mathtt{g}, \ell/\mathtt{l}]$ has no free variables. A *statement context* $S$ is a term derived from the grammar

$$S ::= \diamond \;\mid\; S;\; s.$$

We write $S[s]$ for the statement obtained by substituting a statement $s$ for the unique occurrence of $\diamond$ in $S$, and write $\langle g, \langle \ell, S\rangle\, w, m\rangle\, [s]$ to denote the configuration $\langle g, \langle \ell, S[s]\rangle\, w, m\rangle$.

Figure 2 gives the scheduler-parameterized semantics of simple asynchronous programs as a set of operational steps on configurations. The choice operator $\star$ is used in the Call and Return-Sync rules only as a placeholder for an undetermined value. The Assume rule restricts the set of valid executions: a step is only allowed when the predicated expression $e$ evaluates to true. (This statement—usually confined to intermediate languages—is crucial for our reduction to sequential programs in Section 5.) A task $w'$ in a Post-step is said to be *posted*, and a task $w$ in a Dispatch-step is said to be *dispatched*. We refer to the semantics instantiated by a scheduler $M$ as the *$M$-semantics*. The “natural” bag-semantics (see Scheduler 1) corresponds to the usual asynchronous program semantics [37]. We call the semantics of synchronous programs—i.e., those without **post**-statements—the *synchronous semantics*, which is in fact independent of the scheduler.

A configuration $\langle g, \langle \ell, s; \textbf{return } \star\rangle, \mathsf{empty}\rangle$, where $s$ does not contain **return**-statements, is called *$M$-initial*. An *$M$-execution (to $c_j$)* is a configuration sequence $h = c_1 c_2 \ldots c_j$ where

- $c_1$ is $M$-initial, and
- $c_i \to c_{i+1}$ for $1 \le i < j$.

(The initial statement $s$ begins the execution by posting tasks; if no tasks are posted by $s$, the execution ends when $s$ completes.) We say a configuration $c = \langle g, w, m\rangle$ (alternatively, the global value $g$) is *$M$-reachable* when there exists an $M$-execution to $c$, and is *$M$-final* when in addition $w = \varepsilon$ and $\mathsf{take}(m) = \emptyset$.

For a scheduler $M$ and program $P$, the *$M$-value set on* $P$ is the set of values $\mathtt{g}(c)$ of the global variable g such that $c$ is $M$-final.[2] The $M$-semantics on $P$ is g*-equivalent* to the $M'$-semantics on $P'$ when the $M$-value set on $P$ is equal to the $M'$-value set on $P'$.

Although we have not made task identifiers explicit, we may assume that a set $U \subseteq \mathbb{N}$ of *task identifiers* is defined by an execution $h = c_1 c_2 \ldots c_j$ such that $u \in U$ is the task identifier of a task dispatched[3] by $c_{u-1} \to c_u$. With an execution in mind, there is no ambiguity when referring to the frame or execution of a task by its identifier. We say $u$ is *posted* (resp., *dispatched*) in $h$ when there exists $1 \le i < j$ such that the task identified by $u$ is posted (resp., dispatched) in $c_i \to c_{i+1}$.

## 2.4 Programs with Preemption

The language of *preemptive asynchronous programs* extends the grammar of Figure 1 with the production

$$s ::= \textbf{yield},$$

and the semantics of Figure 2 with the rule

$$\frac{\textsc{Yield} \qquad w' = \langle \ell, S[\textbf{skip}]\rangle\, w}{\langle g, \langle \ell, S[\textbf{yield}]\rangle\, w, m\rangle \to \langle g, \varepsilon, \mathsf{give}(m, w')\rangle},$$

allowing arbitrary task resumptions—not simply single frames—to be added to, and later dispatched from, the task buffer.

## 2.5 Synchronization and Blocking

Our model of preemptive asynchronous programs (i.e., with the **yield**-statement) can express arbitrary synchronization disciplines.

**Example 1** (Locking)**.** Lock-based mutual exclusion can be modeled by adding an additional global variable lock. Critical sections are surrounded with the *lock acquire* operation, encoded by

$$(\textbf{while } \mathtt{lock} = \mathtt{true} \textbf{ do yield}); \mathtt{lock} := \mathtt{true},$$

and the *lock release* operation, encoded by $\mathtt{lock} := \mathtt{false}$. The $\mathsf{blocked}(w, g)$ predicate is defined to hold if and only if $w = \langle \ell, S[\textbf{while } \mathtt{lock} = \mathtt{true} \textbf{ do yield}]\rangle\, w'$ and $\mathtt{lock}(g, \ell) = \mathtt{true}$; i.e., $w$ is waiting for a lock that is in use. Although the mere presence of these primitives does not imply that critical sections are mutually exclusive, ensuring mutual exclusion in their presence is a well-studied, and orthogonal problem [27].

Since exploration deadlocks when blocked tasks are scheduled, we are generally interested in non-blocking schedulers.

**Example 2** (Non-blocking lock scheduling)**.** A deterministic non-blocking scheduler for the lock-based mutual exclusion encoding of Example 1 can be defined by differentiating between *ready* tasks, and tasks *waiting* for the lock; the scheduler must not pick a waiting task when $\mathtt{lock}(g, \ell) = \mathtt{true}$.

[2] In the presence of the **assume**-statement, only the values of completed executions are guaranteed to be valid.

[3] We could also define task identifiers by their tasks' posting-points in $h$.

Skip
$$\frac{}{c[\mathbf{skip}; s] \to c[s]}$$

Assume
$$\frac{\texttt{true} \in e(c)}{c[\mathbf{assume}\ e] \to c[\mathbf{skip}]}$$

If-Then
$$\frac{\texttt{true} \in e(c)}{c[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to c[s_1]}$$

If-Else
$$\frac{\texttt{false} \in e(c)}{c[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to c[s_2]}$$

Loop-Do
$$\frac{\texttt{true} \in e(c)}{c[\mathbf{while}\ e\ \mathbf{do}\ s] \to c[s; \mathbf{while}\ e\ \mathbf{do}\ s]}$$

Loop-End
$$\frac{\texttt{false} \in e(c)}{c[\mathbf{while}\ e\ \mathbf{do}\ s] \to c[\mathbf{skip}]}$$

Assign-Global
$$\frac{g' \in e(g, \ell)}{\langle g, \langle \ell, S[\texttt{g} := e] \rangle\, w, m \rangle \to \langle g', \langle \ell, S[\mathbf{skip}] \rangle\, w, m \rangle}$$

Assign-Local
$$\frac{\ell' \in e(g, \ell)}{\langle g, \langle \ell, S[\texttt{l} := e] \rangle\, w, m \rangle \to \langle g, \langle \ell', S[\mathbf{skip}] \rangle\, w, m \rangle}$$

Call
$$\frac{\ell' \in e(g, \ell)}{\langle g, \langle \ell, S[\mathbf{call}\ x := p(e)] \rangle\, w, m \rangle \to \langle g, \langle \ell', s_p \rangle \langle \ell, S[x := \star] \rangle\, w, m \rangle}$$

Return-Sync
$$\frac{v \in e(g, \ell)}{\langle g, \langle \ell, \mathbf{return}\ e \rangle \langle \ell', S[x := \star] \rangle\, w, m \rangle \to \langle g, \langle \ell', S[x := v] \rangle\, w, m \rangle}$$

Post
$$\frac{\ell' \in e(g, \ell) \qquad w' = \langle \ell', s_p \rangle}{\langle g, \langle \ell, S[\mathbf{post}\ p(e)] \rangle\, w, m \rangle \to \langle g, \langle \ell, S[\mathbf{skip}] \rangle\, w, \mathsf{give}(m, w') \rangle}$$

Dispatch
$$\frac{\langle m', w \rangle \in \mathsf{take}(m, g) \quad \neg\mathsf{blocked}(w, g)}{\langle g, \varepsilon, m \rangle \to \langle g, w, m' \rangle}$$

Return-Async
$$\frac{}{\langle g, \langle \ell, \mathbf{return}\ e \rangle, m \rangle \to \langle g, \varepsilon, m \rangle}$$

Figure 2: The operational semantics of simple asynchronous programs, parameterized by the scheduler $M = \langle D, \mathsf{empty}, \mathsf{give}, \mathsf{take} \rangle$.

# 3 Delay-Bounded Scheduling

To limit the nondeterminism present in a scheduler, and thus the number of executions explored—while at the same time retaining the ability to consider "interesting" executions!—we allow deterministic schedulers to exercise a limited (and parameterized) number of deviations from their deterministic schedules. We define a *delaying scheduler* as a tuple $M = \langle D, \mathsf{empty}, \mathsf{give}, \mathsf{take}, \mathsf{delay} \rangle$, where $D$, empty, give, and take are defined as before for schedulers, and the $\mathsf{delay} : D \times \mathsf{Tasks} \to D$ function is intended to inform the scheduler of task-postponement. The definitions of *deterministic*, *non-blocking*, *pending*, and *schedulable* remain unchanged.

We extend the semantics of Figure 2 with a postponing rule

Delay
$$\frac{\langle m', w \rangle \in \mathsf{take}(m, g)}{\langle g, \varepsilon, m \rangle \to \langle g, \varepsilon, \mathsf{delay}(m', w) \rangle}$$

which we henceforth refer to as a *delay (operation)*, and we say $w$ is *delayed*. Note that the delay operation occurs at the point when tasks are usually dispatched, not the at point when they are posted. We say an execution $h$ is $K$-*delay-bounded* when the number of delay operations in $h$ is at most $K$. A delaying scheduler $M$ is *limit sound* when for any bag-reachable global value $g$, there exists $K \in \mathbb{N}$ such that $g$ is $M$-reachable in a $K$-delay-bounded execution.

Limit soundness has an operational characterization. When $h$ is an execution to $c$, we say $w$ is *blocked* (resp., *pending*) in $h$ when $w$ is blocked (resp., pending) in $c$. A delaying scheduler $M$ is *lossless* when for any $M$-execution $h$ and task $w$ posted[4] in $h$, $w$ is either dispatched, pending, or blocked in $h$; otherwise we say $M$ is *lossy.* A delaying scheduler $M$ is *delay-accessible* when for every reachable configuration $c_1$ with non-blocked, pending task $w$, there exists a sequence $c_1 \to \ldots \to c_j$ of Delay-steps such that $w$ is schedulable in $c_j$.

**Lemma 1.** *A delaying scheduler $M$ is limit sound if $M$ is lossless and delay-accessible.*

Intuitively, a lossless and delay-accessible scheduler can always spend delays to access any task pending in the bag scheduler, since bag-pending tasks are also pending in lossless schedulers. A detailed proof appears in Appendix B.

As a consequence of Lemma 1, our approach will discover any bug, with some delay-bound, using any well-behaved (i.e., lossless & delay-accessible) deterministic scheduler—though the delay-bound required to uncover a given bug depends on the scheduler. Since the number of explored schedules is exponential in the delay-bound, in practice one may want to hand-craft the scheduler to fit a particular program- or bug-domain. Alternatively, it is also possible to run several explorations with different schedulers in parallel with the same (small) delay-bound, or even to choose schedulers at random.

[4] Here we assume $w$ is uniquely identified by its task identifier.

Here we highlight a couple of simple deterministic delaying schedulers to compare with existing exploration techniques, and as a basis for defining practical exploration algorithms.

## 3.1 Round-Robin Scheduling

One simple scheduler cycles through tasks in task-creation order. The scheduler advances to the next task when the current task either (i) completes execution, (ii) yields, and is blocked, or (iii) is delayed.

**Scheduler 3** (The round-robin scheduler)**.** Let $\mathsf{rr}$ be the list-based delaying scheduler $\langle \mathsf{Tasks}^* \times \mathbb{N}, \langle \varepsilon, 0 \rangle, \mathsf{give}, \mathsf{take}, \mathsf{delay} \rangle$, where the $\mathsf{give}(\langle m, i \rangle, w)$ operation is defined by[5]

1. if $w = \langle \ell, s_p \rangle$ for some procedure $p$, then append $w$ to $m$;
2. else ($w$ is a yielding task): insert $w$ into $m$ at position $i$,

the $\mathsf{delay}(\langle m, i \rangle, w)$ operation is defined by

1. insert $w$ into $m$ at position $i$;
2. increment $i$ modulo $|m|$,

and the $\mathsf{take}(\langle m, i \rangle, g)$ operation is defined by

1. if $m(j)$ is blocked in $g$ for all $j \in 0 \dots |m| - 1$, return $\emptyset$;
2. while $m(i)$ is blocked in $g$, increment $i$ modulo $|m|$;
3. remove $w$ from $m$ at position $i$;
4. return $w$.

It is easy to see that $\mathsf{rr}$ is deterministic, lossless, delay-accessible, and non-blocking.

The round-robin scheduler demonstrates that delay-bounded scheduling captures context-bounded scheduling [33].

**Example 3** (Context-bounding)**.** In the $\mathsf{rr}$-scheduler, each delay operation simulates a context-switch to the next task; context-switching to any of $n$ tasks generally requires $n$ successive delay operations. Given a "context-bound" $K$ and a program with $n$ tasks, the set of $nK$-delay-bounded $\mathsf{rr}$-executions contains the set of $K$-context-bounded executions.

From the perspective of limiting scheduling choice, context-bounding does not give a direct bound: the number of choices $nK$ is also dependent on the number of tasks. (In contrast, $K$-delay-bounded exploration requires only $K$ choices.)

## 3.2 Depth-First Scheduling

Another simple scheduler schedules all tasks posted by a given task $u$ before scheduling tasks that were pending when $u$ was dispatched, in a stack-based discipline. This scheduler is particularly appealing since, as we show in Section 5, for any delay-bound $K$ and asynchronous program $P$, we can encode the $K$-delay-bounded depth-first semantics of $P$ compactly as a sequential program!

**Scheduler 4** (The depth-first scheduler)**.** Let $\mathsf{dfs}$ be the stack-based scheduler $\langle \mathsf{Tasks}^* \times \mathsf{Tasks}^* \times \mathsf{Tasks}^*, \langle \varepsilon, \varepsilon, \varepsilon \rangle, \mathsf{give}, \mathsf{take}, \mathsf{delay} \rangle$ — for a scheduler object $\langle q_h, q_r, q_d \rangle$, we call $q_h$, $q_r$ and $q_d$ the *handler-*, *round-*, and *delay-stacks*—where the $\mathsf{give}(\langle q_h, q_r, q_d \rangle, w)$ operation is defined by

1. push $w$ to $q_h$,

the $\mathsf{delay}(\langle q_h, q_r, q_d \rangle, w)$ operation is defined by

1. push $w$ to $q_d$,

and the $\mathsf{take}(\langle q_h, q_r, q_d \rangle, g)$ operation is defined by

1. empty $q_h$ into $q_r$;
2. if $q_r = \varepsilon$ then empty $q_d$ into $q_r$;
3. if $q_r = \varepsilon$ then return $\emptyset$;
4. pop $w$ from $q_r$;
5. return $w$.

Figure 3 shows the $\mathsf{dfs}$-execution of a simple asynchronous program. It is easy to see that $\mathsf{dfs}$ is deterministic, lossless, and delay-accessible, though is not generally non-blocking.

To understand the task-execution order in depth-first scheduling, we view executions as trees, where nodes are tasks, and the parent relation corresponds to the *posting* relation on tasks. Let $h$ be an execution and $U$ a set of task identifiers from $h$. We define the *asynchronous call forest*[6] as the ordered forest $F$ with nodes labeled by the identifiers $u \in U$ of tasks posted in $h$. When $u$ was posted by a task $u'$ in $h$, $u$ is a child of $u'$ in $F$; children are ordered by the order they are posted. We capture yielding by viewing task-resumptions as newly-posted tasks. Note that synchronous calls are not explicit in $F$: the tasks posted along the synchronous execution of $u$ appear directly as children of $u$.

The $\mathsf{dfs}$-execution of a program proceeds in a sequence of "rounds," ending when a $\mathsf{take}$ operation encounters both $q_h$ and $q_r$ empty. With this nomenclature, we schedule each delayed task $w$ in the round $i+1$ following the round $i$ where $w$ is delayed, after all pending non-delayed tasks of round $i$ have been scheduled.

[5] We define the give, delay, and take operations by destructively mutating and returning the scheduler object.

[6] We consider forests rather than trees since more than one task may be posted initially—or as we encounter shortly, at the beginning of each "round" in certain delaying schedulers.

```
var g : $T$
proc $p_1$(var l : $T$)
  $s_1$; post $p_2$ $e_1$;
  $s_2$; post $p_3$ $e_2$; $s_3$
proc $p_2$(var l : $T$) $s_4$
proc $p_3$(var l : $T$) $s_5$
```

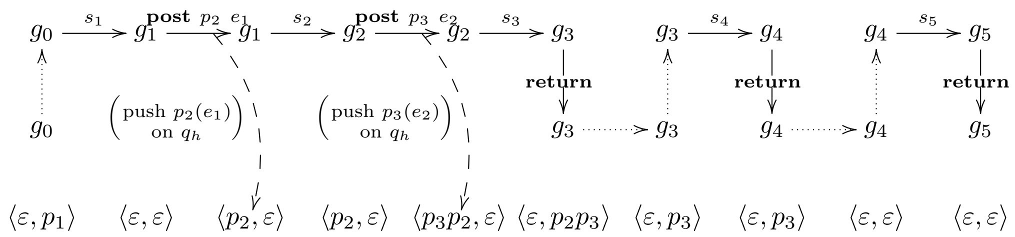

$g_0$
$s_1$
$g_1$
post $p_2$ $e_1$
$g_1$
$s_2$
$g_2$
post $p_3$ $e_2$
$g_2$
$s_3$
$g_3$
$g_3$
$s_4$
$g_4$
$g_4$
$s_5$
$g_5$
return
return
return
$g_0$
push $p_2(e_1)$ on $q_h$
push $p_3(e_2)$ on $q_h$
$g_3$
$g_3$
$g_4$
$g_4$
$g_5$
$\langle \varepsilon, p_1 \rangle$
$\langle \varepsilon, \varepsilon \rangle$
$\langle p_2, \varepsilon \rangle$
$\langle p_2, \varepsilon \rangle$
$\langle p_3 p_2, \varepsilon \rangle$
$\langle \varepsilon, p_2 p_3 \rangle$
$\langle \varepsilon, p_3 \rangle$
$\langle \varepsilon, p_3 \rangle$
$\langle \varepsilon, \varepsilon \rangle$
$\langle \varepsilon, \varepsilon \rangle$

Figure 3: The no-delay dfs-execution of a simple asynchronous program. The scheduler objects $\langle q_h, q_r \rangle$—since there are no delays, $q_d$ is omitted—are given below the corresponding points in the execution which are labeled with the global values. Solid lines indicate procedure control, and dotted lines indicate dispatcher control. Note that the bag scheduler allows one additional execution, where $p_3$ executes before $p_2$.

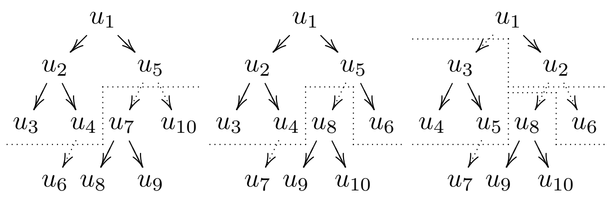


Figure 4: Depth-first traversals of round-partitioned asynchronous call forests with task identifiers $u_1, \ldots, u_{10}$. Arrows indicate task-posting, dotted arrows indicate delays, and the dotted line separates rounds. The traversal-order is $u_1, u_2, \ldots, u_{10}$.

A *round partitioning* of an asynchronous call forest $F$ with nodes $U$ is a labeling $R : U \to \mathbb{N}$ on the nodes of $F$ such that $R(u_1) \leq R(u_2)$ whenever $u_1$ is an ancestor of $u_2$. Given an execution $h$, we can think of the round partitioning as a partition of $F$ into asynchronous call forests $\{F_0, F_1, \ldots\}$ where each task $u$ is in the forest $F_{R(u)}$ corresponding to the round in which it executes. The depth-first scheduler traverses the asynchronous call forest in a round-by-round depth-first preorder (see Figure 4). In this order, yielding tasks $u$ are rescheduled only after the tasks that $u$ has posted before yielding, have been scheduled.

# 4 Delay-Bounded Testing

In concurrency analyses which explicitly enumerate execution schedules (e.g., systematic testing), using delay-bounded deterministic schedulers has a clear scalability advantage over existing bounded exploration approaches. For instance, the number of $p$-preemption-bounded executions [28] of a program using $n$ tasks is exponential in $n$, since a complete execution must apply at least $\max(n, p)$ label changes; the number of $k$-delay-bounded executions does not depend on $n$. Here we demonstrate that although delay-bounding explores much fewer schedules than preemption-bounding (with the same bound), existing bugs caught with $p$ preemptions are also caught with $p$ delays. Furthermore, delay-bounded exploration is able to discover bugs which have not been caught by bounding preemptions—due to the exponential increase in schedules with respect to the number of tasks.

To demonstrate this advantage, we have implemented the delaying round-robin scheduler (Scheduler 3) in the CHESS concurrency testing tool [29] to directly compare preemption- and delay-bounding on three programs: *CCR*, *Futures* (both written by Microsoft product groups), and *Region Ownership* (written by Peter Müller of ETH Zurich). We feel the round-robin scheduler is appropriate for comparison with preemption-bounding since it minimizes preemptions. We observe:

1. Every bug found with $p$ preemptions is also found with $p$ delays.

2. Delay-bounding explores significantly fewer schedules before discovering a buggy execution.

3. Delay-bounding can discover at least one bug that cannot be found with preemption-bounding, under similar time-constraints.

As a systematic testing tool, CHESS repeatedly executes each test case to completion until all possible schedules—for each test case—have been explored. During the exploration, CHESS has no control over the number of tasks created, nor the number of steps taken by the input program. Thus, each test case is expected to drive the program to termination under any schedule.

**CCR** Microsoft's Concurrency and Coordination Runtime provides a concurrent programming model with high-level primitives for data- and work-coordination without the use of explicit threading and synchronization. We evaluate a suite of 42 test cases exercising various parts of CCR. Each test case takes between 82–255 steps to complete, and creates at most 3 tasks, before exhibiting a known bug (found with CHESS); 41 of these bugs were discovered with 1 preemption, and the remaining one with 2 preemptions. The delaying rr-scheduler discovers each of these bugs with the same budget of delays, but fewer schedules. Figure 5 compares the number of schedules explored to discover each

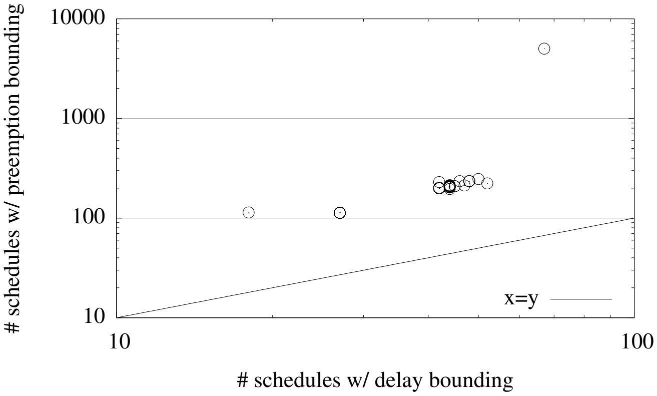


Figure 5: Comparing the number of schedules explored between preemption- and delay-bounding before discovering known bugs in 41 tests of Microsoft's Concurrency and Coordination Runtime (CCR). CHESS found each of these bugs with 1 preemption/delay.

1-preemption bug. Discovering the remaining bug required exploring 8, 895 schedules using 2 preemptions, and only 2, 728 schedules using 2 delays.

**Futures** Microsoft's Futures library provides a synchronization primitive based on a proxy value for a computation whose result is not yet known. We find two previously-discovered (with CHESS) 1-preemption bugs, also with 1 delay. Both the livelock and uncaught exception are exposed after the program has spawned 4 tasks:

| Bug | # steps | # schedules (PB) | # schedules (DB) |
|---|---|---|---|
| Livelock | 1075 | 259 | 45 |
| Exception | 1152 | 1899 | 320 |

**Region Ownership** Peter Müller's region ownership library manages concurrency and coordination based on objects grouped into regions that communicate with each other via asynchronous procedure calls. The library is accompanied by a single test case comprising a one-producer one-consumer system. During testing, at most 5 tasks are created, and at most 284 execution steps taken. Preemption-bounding with 0 and 1 preemptions terminates without finding a bug; with 2 preemptions, CHESS generates 340, 000 schedules over several hours without terminating, after which we manually killed it. Bounding delays to 0, 1, and 2 did not expose any bugs, though a delay-bound of 3 discovered a previously unknown bug after exploring only 132, 507 schedules.

To determine whether this bug can be discovered with 2 preemptions, we re-ran CHESS, focusing preemptions [3] to the methods interrupted in our (delay-bounded) error trace. After exploring 128, 998 schedules, CHESS finishes without discovering the bug. Thus, this is the first bug CHESS has ever found in a real-world program that requires at least 3 preemptions!

# 5 Sequentialization of Depth-First Scheduling

The depth-first scheduling order (i.e., of Scheduler 4) has a rather nice property: the stack of pending tasks used for a depth-first traversal of the asynchronous call forest (see Section 3.2) can be combined with a synchronous program's call stack (i.e., of activation records). In this section we exploit that fact to compactly encode a program's delay-bounded dfs semantics as a sequential program. (A similar encoding is possible for reverse depth-first scheduling—i.e., where each task's children are traversed in reverse order.)

In Section 5.1 we reveal the most basic encoding of no-delay depth-first scheduling for simple asynchronous programs. In Sections 5.2 and 5.3, we orthogonally extend the basic encoding to handle delaying and preemption. To improve clarity we leverage the syntactic extensions of Appendix A, which each reduce to the original syntax of asynchronous programs.

## 5.1 No-Delay Depth-First Scheduling

We begin by defining the function $[\![\cdot]\!]^0_{\mathsf{dfs}}$ which translates a simple asynchronous program $P$ into a synchronous program $[\![P]\!]^0_{\mathsf{dfs}}$ which encodes the no-delay depth-first scheduler:

$$\begin{aligned}
[\![\textbf{var } \mathsf{g} : T\ \vec{H}]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{var } \mathsf{g} : T,\ \bar{\mathsf{g}} : T\ \overrightarrow{[\![H]\!]^0_{\mathsf{dfs}}} \\
[\![\textbf{proc}\ \ p\ (\textbf{var}\ \mathtt{l} : T)\ s]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{proc}\ \ p\ (\textbf{var}\ \mathtt{l} : T)\ [\![s]\!]^0_{\mathsf{dfs}} \\
[\![s_1;\ s_2]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} [\![s_1]\!]^0_{\mathsf{dfs}};\ [\![s_2]\!]^0_{\mathsf{dfs}} \\
[\![x := e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} x := e \\
[\![\textbf{skip}]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{skip} \\
[\![\textbf{assume } e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{assume } e \\
[\![\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{if } e \textbf{ then } [\![s_1]\!]^0_{\mathsf{dfs}} \textbf{ else } [\![s_2]\!]^0_{\mathsf{dfs}} \\
[\![\textbf{while } e \textbf{ do } s]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{while } e \textbf{ do } [\![s]\!]^0_{\mathsf{dfs}} \\
[\![\textbf{call } x := p\ e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{call } x := p\ e \\
[\![\textbf{return } e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{return } e \\
[\![\textbf{post } p\ e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{let } \mathsf{g}_{\mathtt{tmp}} : T = \mathsf{g} \textbf{ in} \\
&\phantom{\overset{\text{def}}{=}} \textbf{let } \bar{\mathsf{g}}_{?} : T\ \ \textbf{in} \\
&\phantom{\overset{\text{def}}{=}} \qquad \mathsf{g} := \bar{\mathsf{g}}; \\
&\phantom{\overset{\text{def}}{=}} \qquad \bar{\mathsf{g}} := \bar{\mathsf{g}}_{?}; \\
&\phantom{\overset{\text{def}}{=}} \qquad \textbf{call } \_ := p\ e; \\
&\phantom{\overset{\text{def}}{=}} \qquad \textbf{assume }\ \mathsf{g} = \bar{\mathsf{g}}_{?}; \\
&\phantom{\overset{\text{def}}{=}} \qquad \mathsf{g} := \mathsf{g}_{\mathtt{tmp}}.
\end{aligned}$$

The initial configuration is translated as

$$[\![\langle g, \langle \ell, s; \textbf{return } e\rangle, m\rangle]\!]^0_{\mathsf{dfs}} \overset{\text{def}}{=} \langle g', \langle \ell, s'; \textbf{return } e\rangle, m\rangle,$$

with the initial global value $g' \overset{\text{def}}{=} \{\mathsf{g} = g, \bar{\mathsf{g}} = \star\}$, and the initial statement $s'$ given by

$$\begin{array}{l} s' \overset{\text{def}}{=} \textbf{let } \bar{\mathsf{g}}_? \textbf{ in} \\ \qquad\qquad \bar{\mathsf{g}} := \bar{\mathsf{g}}_?; \\ \qquad\qquad \llbracket s \rrbracket^0_{\mathsf{dfs}}; \\ \qquad\qquad \textbf{assume } \mathsf{g} = \bar{\mathsf{g}}_?; \\ \qquad \mathsf{g} := \bar{\mathsf{g}}. \end{array}$$

The key mechanism enabling the encoding is the introduction of *guessed global values* in the form of unconstrained symbolic constants $\bar{\mathsf{g}}_?$. These are essentially prophecy variables [1] whose values will be available only later in the sequential execution. Once the appropriate global values are available, the corresponding guesses are suitably constrained.

In the translation, we replace **post**-statements with **call**-statements, and introduce the following variables to ensure the correct global values are observed along the mimicked $\mathsf{dfs}$-execution:

- $\mathsf{g}_{\mathtt{tmp}}$ caches the global value of a *posting* task, so that the *posted* task can observe its future global value in $\mathsf{g}$, and the posting task can resume from its current global value without interference.
- $\bar{\mathsf{g}}_?$ stores the (guessed) global value reached when each posted task completes.
- $\bar{\mathsf{g}}$ stores the (guessed) global value reached after all tasks that have been posted thus far (i.e., that have appeared on the call-stack) have completed.

**Example 4.** The $\mathsf{dfs}$-encoding of the program in Figure 3 is

```
$\textbf{var}\ \mathsf{g} : T,\ \bar{\mathsf{g}} : T$
$\textbf{proc}\ p_1(\textbf{var}\ \mathtt{l} : T)$
  $s_1$;
  $\textbf{let}\ \mathsf{g}_{\mathtt{tmp}} : T = \mathsf{g}\ \textbf{in let}\ \bar{\mathsf{g}}_? : T\ \textbf{in}$       }  $= \llbracket \textbf{post}\ p_2\ e_1 \rrbracket^0_{\mathsf{dfs}}$
    $\mathsf{g} := \bar{\mathsf{g}};\ \bar{\mathsf{g}} := \bar{\mathsf{g}}_?;$
    $\textbf{call}\ \_ := p_2\ e_1;$                                  }  $\mathsf{g}_{\mathtt{tmp}} = g_1, \bar{\mathsf{g}}_? = g_4$
    $\textbf{assume}\ \mathsf{g} = \bar{\mathsf{g}}_?;\ \mathsf{g} := \mathsf{g}_{\mathtt{tmp}};$
  $s_2$;
  $\textbf{let}\ \mathsf{g}_{\mathtt{tmp}} : T = \mathsf{g}\ \textbf{in let}\ \bar{\mathsf{g}}_? : T\ \textbf{in}$       }  $= \llbracket \textbf{post}\ p_3\ e_2 \rrbracket^0_{\mathsf{dfs}}$
    $\mathsf{g} := \bar{\mathsf{g}};\ \bar{\mathsf{g}} := \bar{\mathsf{g}}_?;$
    $\textbf{call}\ \_ := p_3\ e_2;$                                  }  $\mathsf{g}_{\mathtt{tmp}} = g_2, \bar{\mathsf{g}}_? = g_5$
    $\textbf{assume}\ \mathsf{g} = \bar{\mathsf{g}}_?;\ \mathsf{g} := \mathsf{g}_{\mathtt{tmp}};$
  $s_3$
$\textbf{proc}\ p_2(\textbf{var}\ \mathtt{l} : T)\ s_4$
$\textbf{proc}\ p_3(\textbf{var}\ \mathtt{l} : T)\ s_5,$
```

where we assume $s_1$–$s_5$ do not contain **post**-statements. To the right of the braces we indicate the values stored in the let-bound variables $\mathsf{g}_{\mathtt{tmp}}$ and $\bar{\mathsf{g}}_?$, which are the same values used below. This program has the following unique, sequential execution:

$$\begin{tikzcd}
 & g_3 \arrow[r, "s_4"] & g_4 \arrow[d, "\textbf{return}"] & g_4 \arrow[r, "s_5"] & g_5 \arrow[d, "\textbf{return}"] & \\
g_0 \arrow[r, "s_1"] & g_1 \arrow[u, Rightarrow, "\textbf{call}\ p_2\ e_1"] & g_1 \arrow[r, "s_2"] & g_2 \arrow[u, Rightarrow, "\textbf{call}\ p_3\ e_2"] & g_2 \arrow[r, "s_3"] & g_3,
\end{tikzcd}$$

where the double lines indicate part of a $\llbracket \textbf{post} \ldots \rrbracket^0_{\mathsf{dfs}}$-translation (i.e., up to the **call**). Here each global value $g_i$ matches that of the (asynchronous) $\mathsf{dfs}$-execution (of Figure 3). Execution begins with an initial guess for $\bar{\mathsf{g}}$ of $g_3$, i.e., the global value after $p_1$ executes. The latter guessed values for $\bar{\mathsf{g}}$ of $g_4$ and $g_5$ are the global values after the execution of $p_2$ and $p_3$. The final global value is $g_5$.

**Lemma 2.** *Let $P$ be a simple asynchronous program. The synchronous semantics of $\llbracket P \rrbracket^0_{\mathsf{dfs}}$ is $\mathsf{g}$-equivalent to the no-delay $\mathsf{dfs}$-semantics of $P$.*

*Proof sketch.* Let $h$ be a $\mathsf{dfs}$-execution of $P$, and consider the asynchronous call forest $F$ of $h$ with tasks $U = \{u_1, u_2, \ldots u_j\}$. Furthermore, without loss of generality, suppose the dispatch order $\sqsubset$ of tasks in $h$ is $u_1 \sqsubset u_2 \sqsubset \ldots \sqsubset u_j$. Since the $\mathsf{dfs}$-schedule corresponds to a preorder depth-first traversal of the forest $F$, each task $u$'s children $u'$ are executed before any other pending task or task to be posted later in $u$'s execution. In this way, the order $\sqsubset$ corresponds exactly to the execution of *synchronously* executed tasks—i.e., the order in which tasks would execute had they been *called* instead of *posted*—except that the tasks would observe not the semantically-consistent global state at the end of the current task's execution, but an intermediate global state. To ensure that tasks observe the semantically-consistent global state, we rely on the following invariant of the synchronous semantics of $\llbracket P \rrbracket^0_{\mathsf{dfs}}$:

> The value of $\bar{\mathsf{g}}$ at the beginning of execution for each task $u_{i+1}$ is equal to the final value of $\mathsf{g}$ at the end of execution for the task $u_i$ immediately preceding $u_{i+1}$ in the order $\sqsubset$.

Combining the invariant with the coincidence of task-execution order in the $\mathsf{dfs}$-schedule of $P$ and the synchronous program $\llbracket P \rrbracket^0_{\mathsf{dfs}}$, we have the sequence $g_0 g_1 g_2 \ldots g_j$ of global values during dispatch points of $h$ (i.e., when control is given to the dispatcher) is equal to the sequence $\bar{g}_0 \bar{g}_1 \bar{g}_2 \ldots \bar{g}_j$ of global values before the synchronous execution of each task, and after all tasks have executed. Finally, the last statement to be executed in $\llbracket P \rrbracket^0_{\mathsf{dfs}}$ sets the global value to $\bar{g}_j$, ensuring the final global states of $P$ and $\llbracket P \rrbracket^0_{\mathsf{dfs}}$ are equal. □

Note that in general, for an arbitrary deterministic scheduler, algorithmic delay-bounded exploration is not possible. For example, even no-delay scheduling with the round-robin scheduler (Scheduler 3) is undecidable, by reduction from the state-reachability problem for Turing-complete Queue machines. From an automata-theoretic point of view, delay-bounded depth-first scheduling remains decidable since the asynchronous task-buffer is a stack which can be combined with the synchronous activation stack.

## 5.2 Delaying Depth-First Scheduling

The function $[\![\cdot]\!]^K_{\mathsf{dfs}}$ translates a simple asynchronous program $P$ into a synchronous program $[\![P]\!]^K_{\mathsf{dfs}}$ which encodes the $K$-delay bounded depth-first scheduler:

```
$[\![\textbf{var}\ \mathtt{g} : T\ \vec{H}]\!]^K_{\mathsf{dfs}} \overset{\text{def}}{=} \textbf{var}\ \mathtt{g} : T,\ \bar{\mathtt{G}} : T^{K+1},\ \mathtt{k_{delay}} : \mathbb{N}$
                                  $\overrightarrow{[\![H]\!]^K_{\mathsf{dfs}}}$
$[\![\textbf{proc}\ \ p\ (\textbf{var}\ \mathtt{l} : T)\ s]\!]^K_{\mathsf{dfs}} \overset{\text{def}}{=} \textbf{proc}\ \ p\ (\textbf{var}\ \mathtt{l} : T, \mathtt{k_{round}} : \mathbb{N})\ [\![s]\!]^K_{\mathsf{dfs}}$
$[\![\textbf{call}\ x := p\ e]\!]^K_{\mathsf{dfs}} \overset{\text{def}}{=} \textbf{call}\ x := p\ (e, \mathtt{k_{round}})$
$[\![\textbf{post}\ p\ e]\!]^K_{\mathsf{dfs}} \overset{\text{def}}{=} \textbf{let}\ \mathtt{g_{tmp}} : T = \mathtt{g}\ \textbf{in}$
    $\textbf{let}\ \bar{\mathtt{g}}_? : T\ \ \textbf{in}$
        $\textbf{var}\ \mathtt{k} : \mathbb{N} := \mathtt{k_{round}};$
        $\textbf{while}\ \star\ \textbf{and}\ \mathtt{k_{delay}} > 0\ \textbf{do}$
            $\mathtt{k_{delay}} := \mathtt{k_{delay}} - 1;$
            $\mathtt{k} := \mathtt{k} + 1$
        $\mathtt{g} := \bar{\mathtt{G}}\mathtt{[k]};$
        $\bar{\mathtt{G}}\mathtt{[k]} := \bar{\mathtt{g}}_?;$
        $\textbf{call}\ \_ := p\ (e, \mathtt{k});$
        $\textbf{assume}\ \ \mathtt{g} = \bar{\mathtt{g}}_?;$
        $\mathtt{g} := \mathtt{g_{tmp}},$
```

where the omitted statements are translated exactly as in the no-delay translation $[\![\cdot]\!]^0_{\mathsf{dfs}}$. The initial configuration is translated as

$$[\![\langle g, \langle \ell, s; \textbf{return}\ e\rangle, m\rangle]\!]^K_{\mathsf{dfs}} \overset{\text{def}}{=} \langle g', \langle \ell', s'; \textbf{return}\ e\rangle, m\rangle,$$

where the initial global and local values $g'$ and $\ell'$ are

$$\begin{aligned} g' &\overset{\text{def}}{=} \{\mathtt{g} = g, \bar{\mathtt{G}} = \mathtt{[}\star, \star, \ldots\mathtt{]}, \mathtt{k_{delay}} = K\},\ \text{and} \\ \ell' &\overset{\text{def}}{=} \{\mathtt{l} = \ell, \mathtt{k_{round}} = \star\}, \end{aligned}$$

and the initial statement $s'$ is given by

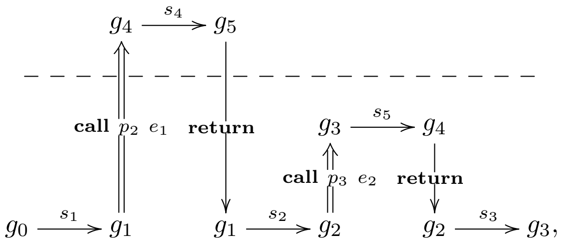
```
$s' \overset{\text{def}}{=} \textbf{let}\ \mathtt{k_{round}} : \mathbb{N} = 0\ \textbf{in}$
    $\textbf{let}\ \bar{\mathtt{G}}_? : T^{K+1}\ \textbf{in}$
        $\bar{\mathtt{G}} := \bar{\mathtt{G}}_?;$
        $[\![s]\!]^K_{\mathsf{dfs}};$
        $\textbf{assume}\ \mathtt{g} = \bar{\mathtt{G}}_?\mathtt{[0]};$
        $\textbf{assume}\ \bar{\mathtt{G}}\mathtt{[0]} = \bar{\mathtt{G}}_?\mathtt{[1]};$
        $\ldots$
        $\textbf{assume}\ \bar{\mathtt{G}}\mathtt{[}K-1\mathtt{]} = \bar{\mathtt{G}}_?\mathtt{[}K\mathtt{]};$
    $\mathtt{g} := \bar{\mathtt{G}}\mathtt{[}K\mathtt{]}.$
```

The $K$-delay-bounded encoding extends the no-delay encoding in two important ways. First, the schedule proceeds in $K+1$ rounds; we must note which round each task executes in, and separately accumulate the perceived global values in each round. Second, each time a task is posted, there is the possibility of delay; we must keep track of how many delays have been spent. To accomplish this, we introduce the following auxiliary variables:

- $\mathtt{g_{tmp}}$ and $\bar{\mathtt{g}}_?$ are used exactly as before, to cache the posting task's observed global value, and store the global value reached at the end of the posted task.
- $\bar{\mathtt{G}}$ is the multi-round extension to $\bar{\mathtt{g}}$: each $\bar{\mathtt{G}}\mathtt{[}i\mathtt{]}$ stores the global value reached after all tasks that have been posted thus far, and are executed in round $i$, have completed.
- $\mathtt{k_{delay}}$ stores the remaining budget of delays.
- $\mathtt{k_{round}}$ indicates which round a given task executes in.
- $\mathtt{k}$ is incremented once per delay of the posted task, then passed as the round-indicator ($\mathtt{k_{round}}$).

Note that a task may be delayed more than once, and we simulate all the delays of a given task instantaneously.

**Example 5.** The 1-delay-bounded $\mathsf{dfs}$-encoding of the program of Figure 3, allows the following (sequential) execution:



where $p_2$ executes before $p_3$. This execution mimics two rounds, separated by a dashed line: $p_3$, though posted second, executes with $p_1$ in the first round, while $p_2$ executes alone in the second. Thus $\bar{\mathtt{G}}\mathtt{[0]}$ takes the values $g_3$, $g_4$, and $\bar{\mathtt{G}}\mathtt{[1]}$ takes the values $g_4$, $g_5$. The final global value is $g_5$.

**Lemma 3.** *Let $P$ be a simple asynchronous program. The synchronous semantics of $[\![P]\!]^K_{\mathsf{dfs}}$ is $\mathtt{g}$-equivalent to the $K$-delay $\mathsf{dfs}$-semantics of $P$.*

*Proof sketch.* As in the proof sketch of Lemma 2, there is a correspondence between the $K$-delay-bounded $\mathsf{dfs}$-execution order of tasks in $P$ and $[\![P]\!]^K_{\mathsf{dfs}}$, except in this case the order is a round-by-round depth-first preorder. Here, we adapt the previous invariant to the $K$-round case:

> The value of $\bar{\mathtt{G}}\mathtt{[}k\mathtt{]}$ at the beginning of execution for each task $u_{i+1}$ of round $k$ is equal to the value of $\mathtt{g}$ at the end of execution for the task $u_i$ immediately preceding $u_{i+1}$ in the order $\sqsubset$.

For adjacently executed tasks $u_i \sqsubset u_{i+1}$ in differing rounds $k_a$ and $k_b$, the resulting global value of $u_i$ is guaranteed to be in $\bar{\mathtt{G}}\mathtt{[}k_a\mathtt{]}$, which is in turn guaranteed to be equal the initial global value $\bar{\mathtt{G}}_?\mathtt{[}k_b\mathtt{]}$ of task $u_{i+1}$. □

$$
\begin{aligned}
[\![\textbf{var } \mathtt{g} : T \vec{H}]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{var } \mathtt{g} : T,\ \bar{\mathtt{g}} : T\ \overrightarrow{[\![H]\!]^0_{\mathsf{dfs}}}\\
[\![\textbf{proc } p\ (\textbf{var } \mathtt{l} : T)\ s]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{proc } p\ (\textbf{var } \mathtt{l}, \bar{\mathtt{g}}_? : T)\ [\![s]\!]^0_{\mathsf{dfs}}\\
[\![\textbf{call } x := p\ e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{call } (x, \bar{\mathtt{g}}_?) := p\ (e, \bar{\mathtt{g}}_?)\\
[\![\textbf{return } e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{return } (e, \bar{\mathtt{g}}_?)\\
[\![\textbf{post } p\ e]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{let } \mathtt{g}_{\mathtt{tmp}} : T = \mathtt{g} \textbf{ in}\\
&\phantom{\overset{\text{def}}{=}} \textbf{let } \bar{\mathtt{g}}'_? : T \textbf{ in}\\
&\phantom{\overset{\text{def}}{=}} \quad \mathtt{g} := \bar{\mathtt{g}};\\
&\phantom{\overset{\text{def}}{=}} \quad \bar{\mathtt{g}} := \bar{\mathtt{g}}'_?;\\
&\phantom{\overset{\text{def}}{=}} \quad \textbf{call } (\_, \bar{\mathtt{g}}'_?) := p\ (e, \bar{\mathtt{g}}'_?);\\
&\phantom{\overset{\text{def}}{=}} \quad \textbf{assume } \mathtt{g} = \bar{\mathtt{g}}'_?;\\
&\phantom{\overset{\text{def}}{=}} \quad \mathtt{g} := \mathtt{g}_{\mathtt{tmp}}\\
[\![\textbf{yield}]\!]^0_{\mathsf{dfs}} &\overset{\text{def}}{=} \textbf{assume } \mathtt{g} = \bar{\mathtt{g}}_?;\\
&\phantom{\overset{\text{def}}{=}} \bar{\mathtt{g}}_? := \star;\\
&\phantom{\overset{\text{def}}{=}} \mathtt{g} := \bar{\mathtt{g}};\\
&\phantom{\overset{\text{def}}{=}} \bar{\mathtt{g}} := \bar{\mathtt{g}}_?
\end{aligned}
$$

Figure 6: The symbolic encoding of no-delay depth-first scheduling with yields extending the symbolic encoding of Section 5.1. The translation of the missing control-flow statements and initial configuration is identical to the yield-free encoding.

## 5.3 Depth-First Scheduling with Preemption

Perhaps surprising is the fact that our symbolic encodings of delaying depth-first schedulers can be extended to preemptive asynchronous programs. The encoding of Figure 6 extends the no-delay depth-first scheduler encoding of Section 5.1 to handle **yield**-statements. The key difference is that the guess ($\bar{\mathtt{g}}_?$) of the post-state of each handler should not be verified only at the end of a handler's execution; instead, the guess is validated if a handler yields, at which point a new guess is made, and the new guess will be validated either at the next yield point—if one exists—or at the end of the handler's execution. To allow multiple guesses throughout a handler's execution, we simply ensure the guess-variable $\bar{\mathtt{g}}_?$ is in scope throughout by making it a parameter to every procedure.

The extension to delaying depth-first scheduling is straightforward; extending to multiple rounds is orthogonal to handling yields, and is done exactly according to the extension of Section 5.2. In particular, occurrences of $\bar{\mathtt{g}}$ are replaced by their multi-round counterparts $\bar{\mathtt{G}}\mathtt{[k]}$, and incrementing the round counters also happens at yield-points. We omit the full definition of $[\![P]\!]^K_{\mathsf{dfs}}$ for preemptive asynchronous programs $P$, since it is redundant.

**Lemma 4.** *Let $P$ be a preemptive asynchronous program. The synchronous semantics of $[\![P]\!]^K_{\mathsf{dfs}}$ is* g*-equivalent to the $K$-delay* dfs*-semantics of $P$.*

## 5.4 Complexity of Depth-First Scheduling

Thus far we have given sequentializations that reduce delay-bounded depth-first semantics to sequential semantics. Since the number of program variables in the resulting sequential program is $\mathcal{O}(K)$, the worst-case complexity of program-state exploration using this reduction (for programs with finite-data domains) is exponential in $K$. What remains is the question of whether the exploration via this reduction is (asymptotically) optimal. Here we find that a sub-exponential algorithm is unlikely. The proofs of these results are quite technical, and can be found in Appendix B.

For the remainder of this section we assume the data-domain of asynchronous programs (i.e., the set Vals) is finite-state. We show that delay-bounded depth-first state-reachability is an NP-complete problem.[7] Though this exploration corresponds to an *under*approximation of the program semantics, the complexity is lower than the precise EXPSPACE-complete explorations of Sen and Viswanathan [37] and Jhala and Majumdar [18] for non-preemptive asynchronous programs. Additionally, our underapproximation has a lower complexity than the PSPACE-hard underapproximation [5] used by Jhala and Majumdar [18]'s algorithm, which corresponds to our (non-preemptive) bounded bag semantics (see Scheduler 2). Note in the case of preemptive programs, the analysis problem is generally undecidable [35].

Interestingly, delay-bounded depth-first exploration has the same NP-complete complexity as context-bounding for a finite number of tasks [26], even though delay-bounded scheduling explores an unbounded number of tasks.

**Problem 1** (Delay-bounded depth-first scheduling)**.** *For a given $K \in \mathbb{N}$,[8] initial configuration $c_0$, and global value $g$ of an asynchronous program $P$, does there exist a $K$-delay-bounded* dfs*-execution to $g$?*

By reduction from the Circuit Satisfiability problem [32], we show our exponential algorithm for delay-bounded depth-first scheduling is likely to be asymptotically optimal.

**Theorem 1.** *Delay-bounded depth-first scheduling is NP-hard.*

To show membership in NP for the non-preemptive case, we give an algorithm that validates a nondeterministically-guessed execution witness to the target global value, given by the sequence of tasks delayed in, and a global value reached at the end of, each round. For a delay-bound $K$, we validate the guess by applying $K+1$ polynomial-time sequential program analyses to sequential encodings of each

[7] Since we are interested in measuring the *scheduling* complexity (rather than complexity arising from program data), we have restricted the program syntax so that a fixed number of variables is in scope at any moment. Indeed, the reachability problem is EXPTIME-complete in the number of variables (even with only a single recursive task), due to the logarithmic encoding of states in the corresponding pushdown system.

[8] We assume the delay-bound $K$ is written in unary.

round of execution—the initial conditions of each round are determined by the delayed tasks and final global state of the previous round. Validation ensures that each round can indeed reach the guessed global value while delaying exactly the guessed delayed tasks.

**Theorem 2.** *Delay-bounded depth-first scheduling for simple asynchronous programs is in NP.*

In fact we can extend the proof of Theorem 2 to the preemptive case. The presence of yields poses an additional technical challenge: a naïve extension of the execution witnesses to record delayed task-*resumptions* does not work, because the resumptions' activation stacks have no bound. We solve this problem essentially by realizing the activation stacks of preempted tasks need not be stored across rounds; instead we may re-execute tasks to recreate their stacks from scratch—the same so-called "lazy" sequentialization technique pioneered by La Torre et al. [22].

**Theorem 3.** *Delay-bounded depth-first scheduling for preemptive asynchronous programs is in NP.*

Thus we achieve tight complexity-bounds on depth-first scheduling.

**Corollary 1.** *Delay-bounded depth-first scheduling is NP-complete.*

# 6 Delay-Bounded Verification

The sequentialization of Section 5 allows any sequential analysis algorithm to be lifted, immediately, to a concurrent analysis algorithm: a sequential analysis of $[\![P]\!]^K_{\mathsf{dfs}}$ is exposed to all concurrent behaviors of $P$ with a $K$-delay-bounded depth-first scheduler (an underapproximation of $P$'s concurrent semantics). The additional implementation effort is minimal since only the source-to-source translation is required.

As a proof-of-concept, we have implemented a source-to-source symbolic encoding of the delaying depth-first scheduler in the Storm [25] concurrent C-program checker. Storm analyzes *closed* concurrent software modules (i.e., each module is closed from below using stubs for external procedures, and closed from above using a test driver with symbolic inputs). While CHESS *concretely* executes a closed concurrent program with a single input vector (see Section 4), Storm *symbolically* verifies a closed concurrent program on a (potentially unbounded) set of input vectors. To analyze the input program precisely, Storm unfolds loops and recursive procedure calls up to a user-provided bound.

Prior to this work, Storm implemented context-bounded verification for programs with a finite number of statically declared tasks. Our experience applying Storm to realistic programs indicated that the theoretical exponential complexity in the number of execution contexts does manifest in practice [25]. Furthermore, many concurrency errors in realistic programs require a large (i.e., $> 3$) number of tasks to be executed, and the number of tasks required to manifest a concurrency error is a lower bound on the number of required contexts. In these cases, context-bounded discovery becomes prohibitively expensive.

In contrast, delay-bounded scheduling discovers these errors with very few delays—typically 1 or 2. The reason depth-first scheduling works well (despite the fact that it is an artificial ordering) is that the majority of these tasks need not participate in any intricate interaction; they simply need to be executed in causal order to reach a program point which manifests the bug. Thus, our implementation of delay-bounded scheduling improves Storm in two important ways: we enable Storm to handle programs with dynamic task-creation, and to find a thus-far elusive class of bugs at low computational cost.

The (end-to-end) implementation works in three phases. In the first phase, we translate a concurrent C program into a concurrent Boogie [11] program—though Boogie was originally intended as an intermediate language for representing the semantics of simple imperative sequential programs, we have extended the syntax and semantics to express concurrent behavior. In the second phase, we transform the concurrent Boogie program into a sequential Boogie program encoding the delay-bounded depth-first semantics. It is noteworthy that our algorithm was very easy to implement with simple, minimal extensions to Storm. Finally, we verify the resulting sequential Boogie program using the triumvirate of field abstraction [25], verification-condition generation [4], and satisfiability-modulo-theory (SMT) solving [10].

We have applied our implementation to symbolic exploration of over 20 preemptible event-driven device drivers with 1K–30K lines of code. In the process we have found 4 previously unknown bugs with a maximum delay-budget of 2; the developers of these drivers have confirmed the accuracy of these bugs. More importantly, with the ability to handle dynamic task-creation, we can precisely model the asynchrony in the device driver's execution environment (e.g., interrupts, deferred procedure calls, timers, driver request cancellation and completion, etc.), thereby qualitatively extending the applicability and precision of Storm.

# 7 Related Work

The programming models considered in this paper have received plenty of attention from researchers interested in stack-based, finite-data abstractions of concurrent programs. The preemptive model has not been so heavily studied, primarily because the reachability problem is known to be undecidable [35]—even with a finite number of tasks—due to interference between multiple stacks. More attention has been paid to the non-preemptive model of asynchronous programs. Sen and Viswanathan [37] introduced the model explicitly to reason about event-driven programs, and showed that control-state reachability is EXPSPACEhard; Ganty and Majumdar [13] tightened this result to show that the problem is EXPSPACE-complete. To combat this high worst-case theoretical complexity, Jhala and Majumdar [18] suggest a scheme that combines an underapproximate and an overapproximate computation of the reachable states.

Context-bounded verification [34, 33] explores only those executions of a concurrent program in which the number of context switches is bounded globally by a user-supplied value. However, the idea of context-bounding does not make intuitive sense for programs with large or unbounded number of tasks. This problem is well-known, and other researchers have proposed various fixes. Atig et al. [2] suggested stratified context-bounding that allows an unbounded number of context switches without sacrificing decidability. La Torre et al. [23] exploit the nondeterministic round-robin scheduling scheme of Lal and Reps [26] to achieve unbounded number of context switches for parameterized concurrent programs.

Exploiting sequential verifiers for concurrent program verification is also an active area. The KISS verifier [34] pioneered this approach by providing a source-to-source transformation from multithreaded programs into sequential programs that underapproximates the set of behaviors of the original program. Lal and Reps [26] achieved a breakthrough by providing the first source-to-source translation that computes a context-bounded underapproximation for any context-bound. This approach also pioneered the idea of guessing future-values and constraining them later at an appropriate control point in the execution; we exploit this idea in our transformation as well. A key weakness of Lal and Reps [26]'s so-called "eager" approach is that control states unreachable in the original concurrent program may be explored in the transformed sequential program; La Torre et al. [22]'s "lazy" technique addresses this weakness by repeatedly re-executing to the control points where guessed values would have been used; Ghafari et al. [15] empirically compared the two approaches in the verification-condition-checking paradigm where, as opposed to model-checking, benefits of laziness are unclear since the eager approach in fact outperforms the lazy one. Kidd et al. [20] have introduced a reduction from concurrent programs with priority-preemptive schedulers to sequential programs, though the construction requires a bound on the number of tasks. Although none of these sequentializations handle dynamic task-creation, La Torre et al. [24] have recently introduced a sequentialization of their parameterized model-checking algorithm [23] which does handle an unbounded number of tasks.

Finally, runtime-schedule "fuzzers" such as ConTest [12] and CalFuzzer [19] introduce delays by adding sleep statements; PCT [7] introduces delays by randomly perturbing task priorities. These techniques share with delay-bounding the ability to scale to many tasks. In fact, a derandomized version of the PCT algorithm provides a scheduling complexity that is independent of the number of program tasks [6]; the mechanism for achieving this scalability is based on their characterization of a bug's *depth* as the minimum number of events that must occur in a certain order to reveal the bug. Bug-depth attempts, like delay-bounding, to canonize the effort required to discover a given bug, but is defined with respect to ordering constraints rather than deviations from a deterministic scheduler.

# 8 Conclusion

We have introduced a canonical characterization of scheduling nondeterminism by considering deterministic schedulers with the ability to delay their next-scheduled task. We demonstrate that delay-bounding is an effective search prioritization strategy for concurrent programs by extending the applicability of existing prioritization techniques. Furthermore, we identify a lower-complexity concurrent analysis problem via delay-bounded depth-first scheduling, and we show that our depth-first delaying schedulers admit practical sequential reductions, allowing us to lift existing sequential analyses to concurrent analyses. Our approach is generally applicable to concurrent programs with dynamic task-creation, and arbitrary preemption and synchronization.

# Acknowledgments

We thank Ahmed Bouajjani, Pierre Ganty, Rupak Majumdar, and Gennaro Parlato for providing helpful insight.

# References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS '09: Proc. 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.

[3] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *TACAS '10: Proc. 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 420–434. Springer, 2010.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 82–87. ACM, 2005.

[5] A. Bouajjani and R. Majumdar. Personal communication, July 2010.

[6] S. Burckhardt and M. Musuvathi. Personal communication, November 2010.

[7] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS '10: Proc. 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178. ACM, 2010.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[9] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers.* O'Reilly Media, Inc., 3$^{\text{rd}}$ edition, 2005.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08: Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[11] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[12] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[13] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010. `http://arxiv.org/abs/1011.0551`.

[14] J. J. Garrett. Ajax: A new approach to web applications, February 2005. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`.

[15] N. Ghafari, A. J. Hu, and Z. Rakamarić. Context-bounded translations for concurrent software: An empirical evaluation. In *SPIN '10: Proc. 17th International Workshop on Model Checking Software*, volume 6349 of *LNCS*, pages 227–244. Springer, 2010.

[16] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186. ACM, 1997.

[17] J. L. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *ASPLOS '00: Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM, 2000.

[18] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL '07: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 339–350. ACM, 2007.

[19] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV '09: Proc. 21st International Conference on Computer Aided Verification*, volume 5643 of *LNCS*, pages 675–681. Springer, 2009.

[20] N. Kidd, S. Jagannathan, and J. Vitek. One stack to run them all: Reducing concurrent analysis to sequential analysis under priority scheduling. In *SPIN '10: Proc. 17th International Workshop on Model Checking Software*, volume 6349 of *LNCS*, pages 245–261. Springer, 2010.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[22] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV '09: Proc. 21st International Conference on Computer Aided Verification*, volume 5643 of *LNCS*, pages 477–492. Springer, 2009.

[23] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV '10: Proc. 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 629–644. Springer, 2010.

[24] S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing parameterized programs, 2010. Under submission.

[25] S. K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *CAV '09: Proc. 21st International Conference on Computer Aided Verification*, volume 5643 of *LNCS*, pages 509–524. Springer, 2009.

[26] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[27] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[28] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455. ACM, 2007.

[29] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI '08:*

*Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX Association, 2008.

[30] W. Oney. *Programming the Microsoft Windows Driver Model.* Microsoft Press, 2nd edition, 2002.

[31] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX '99: Proc. General Track of the USENIX Annual Technical Conference*, pages 199–212. USENIX, 1999.

[32] C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1993.

[33] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS '05: Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 93–107. Springer, 2005.

[34] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI '04: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24. ACM, 2004.

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

[36] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[37] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV '06: Proc. 18th International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 300–314. Springer, 2006.

# A Syntactic Sugar

The following syntactic extensions are reducible to the original syntax of asynchronous programs of Section 2.2. Here we freely assume the existence of various type- and expression-constructors. This does not present a problem since our program semantics does not restrict the language of types nor expressions.

**Multiple types.** Multiple type labels $T_1, \ldots, T_j$ can be encoded by systematically replacing each $T_i$ with the sum-type $T = \sum_{i=1}^{j} T_i$. This allows local and global variables with distinct types.

**Multiple variables.** Additional variables $\mathtt{x_1} : T_1, \ldots, \mathtt{x_j} : T_j$ can be encoded with a single record-typed variable $\mathtt{x} : T$, where $T$ is the record type $\{\mathtt{f_1} : T_1, \ldots, \mathtt{f_j} : T_j\}$, and all occurrences of $\mathtt{x_i}$ are replaced by $\mathtt{x.f_i}$. When combined with the extension allowing multiple types, this allows each procedure to declare any number and type of local variable parameters, distinct from the number and type of global variables.

**Local variable declarations.** Additional (non-parameter) local variable declarations **var** $\mathtt{l}' : T$ to a procedure $p$ can be encoded by adding $\mathtt{l}'$ to the list of parameters, and systematically adding an initialization expression (e.g., the choice expression $\star$, or `false`) to the corresponding position in the list of arguments at each call site of $p$ to ensure that $\mathtt{l}'$ begins correctly (un)initialized.

**Unused values.** Call assignments **call** $x := p\ e$, where $x$ is not subsequently used, can be written as **call** $\_ := p\ e$, where $\_ : T$ is an additional unread local variable, or simpler yet as **call** $p\ e$.

**Let bindings.** Let bindings of the form **let** $x : T = e$ **in** can be encoded by declaring $x$ as a local variable **var** $x : T$ immediately followed by an assignment $x := e$. This construct is used to explicate that the value of $x$ remains constant once initialized. The binding **let** $x : T$ **in** is encoded by the binding **let** $x : T = \star$ **in**, where $\star$ is the choice expression.

**Tuples.** Assignments $(x_1, \ldots, x_j) := e$ to a tuple of variables $x_1, \ldots, x_j$ are encoded by the sequence **let** $\mathtt{r} : \{\mathtt{f_1} : T_1, \ldots, \mathtt{f_j} : T_j\} = e$ **in** $x_1 := \mathtt{r.f_1}; \ldots; x_j := \mathtt{r.f_j}$, where $\mathtt{r}$ is a fresh variable. A tuple expression $(x_1, \ldots, x_j)$ occurring in a statement $s$ is encoded as **let** $\mathtt{r} : \{\mathtt{f_1} : T_1, \ldots, \mathtt{f_j} : T_j\} = \{\mathtt{f_1} = x_1, \ldots, \mathtt{f_j} = x_j\}$ **in** $s[\mathtt{r}/(x_1, \ldots, x_j)]$, where $\mathtt{r}$ is a fresh variable, and $s[e_1/e_2]$ replaces all occurrences of $e_2$ in $s$ with $e_1$. When a tuple-element $x_i$ on the left-hand side of an assignment is unneeded (e.g., from the return value of a **call**), we may replace the occurrence of $x_i$ with the $\_$ variable—see the "unused values" desugaring.

**Arrays.** Finite $T^j$-arrays with $j$ elements of type $T$ can be encoded as records of type $T' = \{\mathtt{f_1} : T, \ldots, \mathtt{f_j} : T\}$, where $\mathtt{f_1}, \ldots, \mathtt{f_j}$ are fresh names. Occurrences of terms `a[i]` are replaced by $\mathtt{a.f_i}$, and array-expressions $[e_1, \ldots, e_j]$ are replaced by record-expressions $\{\mathtt{f_1} = e_1, \ldots, \mathtt{f_j} = e_j\}$.

# B Proofs to Lemmas & Theorems

**Lemma 1.** *A delaying scheduler $M$ is limit sound if $M$ is lossless and delay-accessible.*

*Proof.* First, note that if $M$ can mimic every `bag`-order of task dispatch (e.g., by spending delays), then the set of $M$-reachable global values contains the set of `bag`-reachable values. We will show that any lossless and delay-accessible scheduler can mimic any bag-execution using a finite delay-budget.

Let $h = \vec{c}_0\vec{c}_1\vec{c}_2\ldots\vec{c}_n$ be a bag-execution to $g$ with exactly $n$ DISPATCH steps $\vec{c}_0 \to \vec{c}_1 \to \ldots \to \vec{c}_n$ scheduling tasks $w_1, \ldots, w_n$. By induction on $n$, we construct from $h$ an $M$-execution $h' = \vec{c}_0\vec{c_1}'\vec{c}_1\vec{c_2}'\ldots\vec{c}_n$ to $g$: just before each dispatch to $w_i$ in $h$, add $k_i$ successive delay steps through $\vec{c_i}'$: $c'_{i,1} \to \ldots \to c'_{i,k_i} \to \vec{c}_i$ such that $w_i$ is schedulable in $c'_{i,k_i}$. Note that $w_i$ is not blocked in $\vec{c}_0\vec{c_1}'\ldots\vec{c}_{i-1}$, since $w_i$ is dispatched in the bag-execution, nor is $w_i$ blocked in $\vec{c}_0\vec{c_1}'\ldots\vec{c}_{i-1}\vec{c_i}'$, since $\vec{c_i}'$ does not change the global state. In fact, $w_i$ is pending in $\vec{c}_0 c\vec{s_1}'\ldots\vec{c}_{i-1}$, and thus in $\vec{c}_0\vec{c_1}'\ldots\vec{c_i}'$, since $M$ is lossless. Since $M$ is accessible, $|\vec{c_i}'| = k_i$ is finite. As $n$ tasks are scheduled, $k_1 + \ldots + k_n \in \mathbb{N}$ total delays suffice to ensure $h'$ is an $M$-execution to $g$. □

**Theorem 1.** *Delay-bounded depth-first scheduling is NP-hard.*

*Proof.* We show that the Circuit Satisfiability problem [32] is polynomial-time reducible to delay-bounded scheduling.

Let $C$ be a Boolean circuit with wires $W$, gates $G$, inputs $I$, and an output wire $w_0 \in W$. Without loss of generality, assume that each gate $g \in G$ is connected to two input wires and two output wires, and each input $h \in I$ is connected to two wires. The Circuit Satisfiability problem asks if there exists a valuation to the inputs $I$ which makes the value on wire $w_0$ true.

We construct a simple asynchronous program $P$ (making use of the syntactic extensions of Appendix A) as follows. Let

$$U = \{\texttt{wire} : [1, |W|], \texttt{active} : \mathbb{B}, \texttt{val} : \mathbb{B}\}, \text{ and}$$
$$T = \{\texttt{one} : U, \texttt{two} : U\}$$

be record types. $P$ declares a global variable $\texttt{g} : T$, a procedure $p_g$ for each gate $g \in G$ and a procedure $p_h$ for each input $h \in I$, as well as a procedure for setting wire values

```
proc set(var l : $T$)
    if $\star$ then g.one := l.one else g.two := l.one
```

and a procedure for retrieving wire values

```
proc get(var l : $T$)
    assume g.one.active and g.one.wire = l.one.wire;
    assume g.two.active and g.two.wire = l.two.wire;
    g.one.active := false;
    g.two.active := false;
    return g.
```

When $h \in I$ is connected to wires $w_1$ and $w_2$, we declare the procedure

```
proc $p_h$(var l : $T$)
    post set({ one = { wire = $w_1$, active = true,
                       val = $\star$},
               two = $\star$});
    post set({ one = { wire = $w_2$, active = true,
                       val = $\star$},
               two = $\star$}),
```

and when $g \in G$ is connected to input wires $w_1$, $w_2$ and output wires $w_3$, $w_4$, and computes the function $f$, we declare the procedure

```
proc $p_g$(var l : $T$)
    l.one.wire := $w_1$;
    l.two.wire := $w_2$;
    l := get(l);
    post set({ one = { wire = $w_3$, active = true,
                       val = $f$(l.one.val, l.two.val)},
               two = $\star$});
    post set({ one = { wire = $w_4$, active = true,
                       val = $f$(l.one.val, l.two.val)},
               two = $\star$}).
```

The initial statement of $P$ then posts one of each input-gate procedure $p_h$ and one of each gate-procedure $p_g$, each with an arbitrary parameter value (these procedures all ignore the initial value of their local variable). Note that the maximum number $K$ of pending tasks is $3(|G| + |I|)$, i.e., the number of inputs, gates, and wires outgoing from each.

Clearly the size of $P$ is polynomial in the size of $C$: the number of procedures is $\mathcal{O}(|G| + |I|)$, and the global storage is $\mathcal{O}(\log |W|)$.

The program simulates the circuit $C$ by evaluating each gate $g$ one-by-one based on a schedule such that $g$'s input wires are active exactly when procedure $p_g$ is scheduled to execute. This is made possible since the *setting* of each input wire of $p_g$ is done as a task, which in turn can be scheduled immediately before $p_g$. Such an execution is explored with at most $K^2$ delays by any scheduler that can access any of $K$ pending tasks with $K$ delays, e.g., the delaying depth-first scheduler.

We then ask if there is a reachable global state where $w_0$ is active, and has value true: i.e., if $\texttt{g.one.wire} = w_0$, $\texttt{g.one.active} = \texttt{true}$, and $\texttt{g.one.val} = \texttt{true}$. If such a state is reachable, then it is not difficult to see $C$ is satisfiable. On the other hand, if the circuit is satisfiable, then there exists an execution with $K$ or fewer delays that reaches a target state, since any possible circuit evaluation order is considered using $K$ delays. □

**Theorem 2.** *Delay-bounded depth-first scheduling for simple asynchronous programs is in NP.*

*Proof.* Let $K$ be the delay-bound, and $\hat{g}$ the target global state of program $P$. We prove membership in NP with an oracle producing a polynomial-sized depth-first scheduling witness of an execution to $\hat{g}$ which we verify in polynomial-time. Our proof uses the intuition of the delay-bounded depth-first symbolic encodings of Section 5.

Without loss of generality, suppose $\hat{g}$ is reachable using exactly $K$ delays. We number the delays from $[1, K]$ in their order of occurrence, and guess the initial global values $\bar{g}_i$ of each round $i$, along with the following $K$-length sequences:

$\mathsf{round} \in [0, K]^K$ indicating the round a given delay occurs in, where $\mathsf{round}(u_1) \leq \mathsf{round}(u_2)$ when $u_1 < u_2$, and

$\mathsf{task} \in (\mathsf{Procs} \times \mathsf{Vals})^K$ indicating which tasks are delayed.

Given these sequences, let $d_i$ (resp., $d'_i$) denote the first (resp., last) delay that occurs in round $i$; when no delays occur in round $i$, we set $d_i = d'_i = K$. (Note that when no delays occur in round $i$, there will be no tasks to delay in the subsequent rounds, so $d_j = d'_j = K$ for all $j \in [i, K]$.)

We must validate the following:

1. The initial global value $\bar{g}_i$ of each round $i > 0$ is reached in (the conclusion of) round $i-1$,
2. the target state $\hat{g}$ is reached in round $K$, and
3. the sequence $u_1 u_2 \dots u_j$ of pending tasks at the beginning of each round $i$ is exactly the sequence of *delayed* tasks of the previous round, i.e., $\{u_1, \dots, u_j\} = \{u \in [1, K] : \mathsf{round}(u) = i - 1\}$.

We validate these conditions by separately analyzing sequential programs $P_0, \dots, P_K$ derived from $P$, such that each $P_i$ mimics the execution of round $i$ according to the guessed execution witness to $\hat{g}$. In each $P_i$, we introduce additional global variables $\bar{\mathsf{g}} \colon T$ and $\mathtt{D} \colon [0, K]$, and replace each statement **post** $p\ e$ with the translation (denoted $[\![$**post** $p\ e]\!]$):

```
if $\star$ and $\langle p, e\rangle = \mathsf{task}(\mathtt{D}+1)$ then
    $\mathtt{D} := \mathtt{D} + 1$
else
    let $\mathsf{g}_{\mathtt{tmp}} : T = \mathsf{g}$ in
    let $\bar{\mathsf{g}}_? : T$ in
        $\mathsf{g} := \bar{\mathsf{g}}$;
        $\bar{\mathsf{g}} := \bar{\mathsf{g}}_?$;
        call $\_ := p\ e$;
        assume $\mathsf{g} = \bar{\mathsf{g}}_?$;
        $\mathsf{g} := \mathsf{g}_{\mathtt{tmp}}$.
```

From an initial configuration $\langle g, \langle \ell, s; \textbf{return}\ \star\rangle, \mathsf{empty}\rangle$ of $P$, the initial configuration of each $P_i$ is $\langle g_i, \langle \ell, s_i; \textbf{return}\ \star\rangle, \mathsf{empty}\rangle$, where $g_i$ is the record-value

$$\begin{array}{ll} \underline{\text{when } i = 0} & \underline{\text{when } 0 < i \leq K} \\ \{\mathsf{g} = g, \bar{\mathsf{g}} = g, \mathtt{D} = 0\} & \{\mathsf{g} = \bar{g}_i, \bar{\mathsf{g}} = \bar{g}_i, \mathtt{D} = 0\}, \end{array}$$

and $s_i$ is the statement

$$\begin{array}{ll} \underline{\text{when } i = 0} & \underline{\text{when } 0 < i \leq K} \\ & [\![\textbf{post}\ \mathsf{task}(d_{i-1})]\!];\ \dots \\ s; & \quad \dots;\ [\![\textbf{post}\ \mathsf{task}(d'_{i-1})]\!]; \\ \textbf{assume}\ \mathtt{D} = d'_i - d_i; & \textbf{assume}\ \mathtt{D} = d'_i - d_i; \\ \mathsf{g} := \bar{\mathsf{g}} & \mathsf{g} := \bar{\mathsf{g}}, \end{array}$$

where $[\![\cdot]\!]$ applies the same translation of **post** described just above. Note that in each $P_i$, the statement $s_i$ ensures that the pending task-handlers of the subsequent round have been delayed, simply by ensuring that the number of delayed handlers—stored in the variable $\mathtt{D}$—matches the number round-$i$ delays ($d'_i - d_i$) of the execution witness; $\mathtt{D}$ is incremented nondeterministically at the point where the next-expected delayed task was supposed to have been posted (see the translation $[\![$**post** $p\ e]\!]$).

For each of the resulting sequential programs $P_i$—each of size $\mathcal{O}(P)$—reachability of the global value $\mathsf{g} = \bar{g}_{i+1}$ (or $\mathsf{g} = \hat{g}$ for $i = K$) is decidable in polynomial time [36]. □

**Theorem 3.** *Delay-bounded depth-first scheduling for preemptive asynchronous programs is in NP.*

*Proof.* Let $K$ be the delay-bound, and $\hat{g}$ the target global state of a program $P$. We prove membership in NP using an oracle producing a polynomial-sized witness of an execution to $\hat{g}$, which we verify in polynomial-time. Our proof uses the intuition of the delay-bounded depth-first scheduler symbolic encoding of Figure 6.

Without loss of generality, suppose $\hat{g}$ is reachable using exactly $K$ delays. We number the delays from $[1, K]$ in their order of occurrence, and guess the initial global value $\bar{g}_i$ of each round $i$, along with the following $K$-length sequences:

$\mathsf{round} \in [0, K]^K$, indicating the round a given delay occurs in, where $\mathsf{round}(u_1) \leq \mathsf{round}(u_2)$ when $u_1 < u_2$,

$\mathsf{task} \in (\mathsf{Procs} \times \mathsf{Vals})^K$ identifying which tasks are delayed,

and to witness the execution of tasks spanning multiple rounds (a la yield),

$\mathsf{first} \in [0, K]^K$ is the index of this task's first delay (0 if this is the first),

$\mathsf{next} \in [0, K]^K$ is the index of this task's next delay (0 if this is the last)

$\mathsf{delayed} \in ([1, K]^*)^K$ indicates the delays that occur within this task's execution in the current round—i.e., delays of tasks that are transitively posted by this task—and

$\mathsf{g\_suspend}, \mathsf{g\_resume} \in \mathsf{Vals}^K$ indicates the global values this task observes upon yielding and resuming.

Given these sequences, let $d_i$ (resp., $d'_i$) denote the first (resp., last) delay that occurs in round $i$; when no delays occur in round $i$ we set $d_i = d'_i = K$. (Note that when no delays occur in round $i$, there will be no tasks to delay in the subsequent rounds, so $d_j = d'_j = K$ for all $j \in [i, K]$.)

We are obliged to validate the following

1. the initial global value $\bar{g}_i$ of each round $i > 0$ is reached in (the conclusion of) round $i-1$,
2. the target state $\hat{g}$ is reached in round $K$,
3. the sequence $u_1 u_2 \ldots u_j$ of pending tasks at the beginning of each round $i$ is exactly the sequence of *delayed* tasks of the previous round, i.e., $\{u_1, \ldots, u_j\} = \{u \in [1, K] : \mathsf{round}(u) = i - 1\}$.

We validate these conditions by separately analyzing sequential programs $P_0, \ldots, P_K$ derived from $P$, such that each $P_i$ mimics the execution of round $i$ according to the guessed execution witness to $\hat{g}$. Since tasks may span multiple rounds, analyzing each round in isolation is tricky: intuition tells us that tasks' local state (i.e., activation stacks) should be preserved (i.e., provided in our execution witness) from one round to the next; however, since each task's local state is unbounded, preserving it would prevents us from achieving a polynomially-sized execution witness. To avoid this problem, we follow the replay approach of La Torre et al. [22]: instead of remembering tasks' local state between rounds, we recreate the local state by re-executing the task from its beginning up until the point where it is suspended in a given round $i$. When we re-encounter the points where the task was suspended (i.e., with the same global state) in a previous round $j < i$, we instantaneously update the global state to reflect the point where the task is resumed.

Still, there is another complication. In La Torre et al. [22]'s setting, only effects to the values of global variables are observed by other tasks; i.e., the "summary" of a task-segment could be captured solely by two global valuations: the initial and final global valuations a task has seen in a given round. In our setting there is an additional effect to witness: tasks, posted by the current, that are delayed until the next round. Thus an re-execution which reaches the value $g$ and delays tasks $\vec{p}_1$ is not a faithful replay of an execution that reaches $g$ and delays $\vec{p}_2 \neq \vec{p}_1$, because the pending tasks of the following round would differ.

Given these additional concerns, we must also to validate for each delayed task $u$ (identifying here with its delay) whose execution spans multiple rounds:

4. that $u$ reaches the global value $\mathsf{g\_suspend}(u)$ during round $\mathsf{round}(u)$ and resumes from the global value $\mathsf{g\_resume}(u)$ during the next round $\mathsf{round}(u) + 1 = \mathsf{round}(\mathsf{next}(u))$, while at the same time,
5. the delayed tasks of round $\mathsf{round}(u)$ which are posted by $u$ (and transitively by tasks that $u$ posts) match the subsequence $\mathsf{delayed}(u)$ of delayed tasks in round $\mathsf{round}(u)$.

To achieve all of this, in each $P_i$, we introduce the following global variables:

$\bar{\mathsf{g}}$: $\mathsf{Vals}$ stores a virtual global-state,

$\mathtt{D}$: $[0, K]$ counts the delays used in round $i$,

$\mathtt{Re}$: $[0, K]$ stores the index of the currently replaying task,

$\mathtt{R}$: $[0, K]$ stores the current round (during replay, $\mathtt{R} < i$), and

$\mathtt{D_s}$: $([0, K])^K$ counts per-round the delays used in a replay.

Additionally, each procedure takes and returns an additional parameter $\bar{\mathsf{g}}_?$ (see Figure 6) so that the guessed global value is in scope at all possible yield points.

We derive $P_i$ from $P$ by replacing each statement **post** $p\ e$ with the translation (denoted $[\![\textbf{post}\ p\ e]\!]$):

```
if $\star$ and $\mathtt{Re} > 0$
    and $\langle p, e\rangle = \mathsf{task}(\mathsf{delayed}(\mathtt{Re})(\mathtt{D_s}[\mathtt{R}]))$ then
        $\mathtt{D_s}[\mathtt{R}] := \mathtt{D_s}[\mathtt{R}] + 1$;
        if $\mathtt{R} = i$ then
            assume $\langle p, e\rangle = \mathsf{task}(\mathtt{D} + 1)$;
            $\mathtt{D} := \mathtt{D} + 1$;
else if $\star$ and $\mathtt{R} = i$ and $\langle p, e\rangle = \mathsf{task}(\mathtt{D} + 1)$ then
    $\mathtt{D} := \mathtt{D} + 1$;
else
    let $\mathsf{g}_{\mathtt{tmp}} : T = \mathsf{g}$ in
    let $\bar{\mathsf{g}}'_? : T$ in
        $\mathsf{g} := \bar{\mathsf{g}}$;
        $\bar{\mathsf{g}} := \bar{\mathsf{g}}'_?$;
        call $(\_, \bar{\mathsf{g}}'_?) := p(e, \bar{\mathsf{g}}'_?)$;
        assume $\mathsf{g} = \bar{\mathsf{g}}'_?$;
        $\mathsf{g} := \mathsf{g}_{\mathtt{tmp}}$,
```

and each statement **yield** with the translation:

```
assume $\mathsf{g} = \bar{\mathsf{g}}_?$;
$\bar{\mathsf{g}}_? := \star$;
if $\star$ and $\bar{\mathsf{g}} = \mathsf{g\_suspend}(\mathtt{Re})$
    and $\mathtt{D_s}[\mathtt{R}] = |\mathsf{delayed}(\mathtt{Re})|$ then
        $\mathtt{R} := \mathtt{R} + 1$;
        if $\mathtt{R} \leq i$ then
            $\bar{\mathsf{g}} := \mathsf{g\_resume}(\mathtt{Re})$;
            $\mathtt{Re} := \mathsf{next}(\mathtt{Re})$;
        else
            return$^*$; [9]
$\mathsf{g} := \bar{\mathsf{g}}$;
$\bar{\mathsf{g}} := \bar{\mathsf{g}}_?$.
```

From an initial configuration $\langle g, \langle \ell, s; \textbf{return } \star \rangle, \mathsf{empty} \rangle$ of $P$, the initial configuration of each $P_i$ is $\langle g_i, \langle \ell, s_i; \textbf{return } \star \rangle, \mathsf{empty} \rangle$, where $g_i$ is the record-value

$$\begin{array}{l}
\underline{\text{when } i = 0} \\
\{\mathsf{g} = g, \bar{\mathsf{g}} = g, \mathtt{D} = 0, \mathtt{Re} = 0, \mathtt{R} = 0, \mathtt{D}_\mathtt{s} = [0, \ldots, 0]\} \\
\underline{\text{when } 0 < i \leq K} \\
\{\mathsf{g} = \bar{g}_i, \bar{\mathsf{g}} = \bar{g}_i, \mathtt{D} = 0, \mathtt{Re} = 0, \mathtt{R} = i, \mathtt{D}_\mathtt{s} = [0, \ldots, 0]\},
\end{array}$$

where and $s_i$ is the statement

$$\begin{array}{ll}
\underline{\text{when } i = 0} & \underline{\text{when } 0 < i \leq K} \\
 & [\![\textbf{post } \mathsf{task}(d_{i-1})]\!]^+; \ \ldots \\
s; & \quad \ldots; \ [\![\textbf{post } \mathsf{task}(d'_{i-1})]\!]^+; \\
\textbf{assume } \mathtt{D} = d'_i - d_i; & \textbf{assume } \mathtt{D} = d'_i - d_i; \\
\mathsf{g} := \bar{\mathsf{g}} & \mathsf{g} := \bar{\mathsf{g}},
\end{array}$$

where $[\![\textbf{post } \mathsf{task}(u)]\!]^+ =$

$$\begin{cases}
[\![\textbf{post } \mathsf{task}(u)]\!] & \text{if } \mathsf{first}(u) = \mathsf{next}(u) = 0 \\
\begin{array}{l}
\mathtt{R} := 0; \\
\mathtt{D}_\mathtt{s} := [0, \ldots, 0]; \\
\mathtt{Re} := \mathsf{first}(u); \\
[\![\textbf{post } \mathsf{task}(u)]\!]; \\
\textbf{assume } \mathtt{R} := i + 1; \\
\mathtt{R} := i;
\end{array} & \text{otherwise}
\end{cases}$$

and $[\![\cdot]\!]$ applies the same translation of **post** described just above. Note that in each $P_i$, the statement $s_i$ ensures that the pending task-handlers of the subsequent round have been delayed, simply by ensuring that the number of delayed handlers—stored in the variable $\mathtt{D}$—matches the number round-$i$ delays $(d'_i - d_i)$ of the execution witness; $\mathtt{D}$ is incremented nondeterministically at the point where the next-expected delayed task was supposed to have been posted (see the translation $[\![\textbf{post } p \ e]\!]$). Similarly, $s_i$ ensures the correct sequence of delays is observed in replayed task segments by checking that the variable $\mathtt{D}_\mathtt{s}[j]$ is incremented $|\mathsf{delayed}(\mathtt{Re})|$ times, for each $0 < j \leq i$; though technically, $s_i$ performs this check indirectly by ensuring the variable $\mathtt{R}$, which is incremented only when the correct value of $\mathtt{D}_\mathtt{s}[j]$ is reached, is incremented once per replayed round. Additionally, ensuring $\mathtt{R} = i + 1$ also guarantees that the correct global values $(\mathsf{g\_suspend}(\mathtt{Re}))$ were reached in each of the replayed rounds.

For each of the resulting sequential programs $P_i$—each of size $\mathcal{O}(P)$—reachability of the global value $\mathsf{g} = \bar{g}_{i+1}$ (or $\mathsf{g} = \hat{g}$ for $i = K$) is decidable in polynomial time [36]. □

[9] Here we assume the syntax of programs contains a **return**$^*$ statement whose semantics is to immediately return control to the bottom-most activation frame. This behavior can be simulated by inserting **return** -statements before every statement (except the statement in the initial frame), guarded by an additional global variable.