Asynchronous Programs with Prioritized Task-Buffers

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

Akash Lal
Microsoft Research
Bangalore, India
akashl@microsoft.com

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

January 1, 2012

Technical Report
MSR-TR-2012-1

Formal programming models are designed to identify and isolate key features in programs. For example, pushdown systems are a natural (and popular) model for sequential recursive programs that isolate the call-return semantics of procedure calls. Isolating features allows an analysis to focus on the key challenges. For concurrent programs, the model of multithreading can sometimes be too general. Consequently, previous work [26] has proposed a model of asynchronous programming where tasks (unit of computation) can be posted to a task buffer and are then executed in a nondeterministically chosen, but serial, order from the buffer.

Guided by real-world applications of asynchronous programming, we propose a new model that enriches the asynchronous programming model by adding two new features: multiple task-buffers and multiple task-priority levels. In our model, tasks can reside in multiple buffers and are served in priority order. Our model allows non-serial execution: tasks with higher priority can preempt tasks with a lower priority, and tasks obtained from different buffers can freely interleave with each other. Modeling these features allows analysis algorithms to detect otherwise uncaught programming errors in asynchronous programs due to inter-buffer interleaving and task-interruption, while correctly ignoring false errors due to infeasible out-of-priority-order executions.

We also give an algorithm for analyzing this new model. Given bounds $K_1, K_2 \in \mathbb{N}$ that restrict inter-buffer task interleaving and intra-buffer task reordering, we give a code-to-code translation from programs in our model to sequential programs, which can then be analyzed by off-the-shelf sequential-program analysis tools. For any given parameter values, the sequential program encodes a subset of possible program behaviors, and in the limit as both parameters approach infinity, the sequential program encodes all behaviors. We demonstrate the viability of our technique by experimenting with a prototype implementation. It is competitive with state-of-the-art verification tools for concurrent programs. Our prototype is able to correctly identify programming errors in simplified asynchronous Windows device driver code, while ignoring infeasible executions.

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

1. Introduction

Users of interactive computer systems expect little or no latency in an application's ability to react to their actions. For instance, when interacting with a graphical user interface (GUI), one expects an instant reaction for each mouse click and key stroke, despite the long running computation the application may be performing. Similarly, one expects a web server to reply to each HTTP request immediately, despite the thousands of concurrent requests the server may be handling. To ensure such low-latency behavior, the modern operating systems on which these applications run provide mechanisms to break and parallelize applications' sequential control flow. Hardware events initiate software handlers which interrupt the currently-executing process, e.g., ensuring each keystroke is communicated to the application immediately. In multi-processing systems, logically disjoint tasks are executed in parallel, e.g., to divide the processing of distinct HTTP connections across several cores or processors.

Traditionally such reactive software systems have been designed as shared-memory multi-process or multi-threaded programs. Whether executing on a single or across multiple processors, a collection of software threads—each essentially behaving as recursive sequential programs—execute concurrently, interleaving their read and write accesses to shared memory. Though simple to state, such a concurrent programming model is complex due to the many possible intricate interactions between threads. Such complexity is troublesome for the designer who must predict and prevent undesirable thread interleavings by adding synchronization such as atomic locking instructions. This job is particularly difficult given that over-synchronizing, e.g., by protecting all transactions by a single global lock, hampers reactivity and destroys opportunities for parallelization. Furthermore, the non-deterministic nature of such interleaving semantics is the root cause of *Heisenbugs*, i.e., programming errors that manifest themselves rarely, and are often very difficult to reproduce and repair.

In reaction to the difficulty of multi-threaded programming, reactive software systems designed according to the asynchronous programming model [9, 11, 26] have gained much traction. An *asynchronous program* divides cumulative program behavior into short-running tasks. Each task behaves essentially as a recursive sequential program, which in addition to accessing a memory shared by all tasks can *post* new tasks to *task-buffers* for later execution. Tasks from each buffer execute serially, one after the other: when one task completes, another task is taken from the buffer and run to completion. To program a reactive system the designer simply must ensure that no single task executes for too long to prevent other, possibly more urgent, tasks from executing. Figure 1 illustrates the general architecture of asynchronous programs.

Due to the relative simplicity, asynchronous programming is becoming a particularly appealing way to implement reactive systems. Asynchronous programming has seen widespread adoption in recent years by desktop applications, servers, and embedded systems alike. The Javascript engines of modern web browsers [10], Grand Central Dispatch in MacOS and iOS [2], Linux's work-queues [27], asynchrony in .NET [20], and deferred procedure calls in the Windows kernel [21] are all based on asynchronous programming. Even in the single-processing setting (i.e., without any parallelism) asynchrony frameworks such as Node.js [6] are becoming widely used to design extremely scalable (web) servers.

Despite the relative simplicity of asynchronous programming, concurrent programming errors are still possible. Since programmers are often required to restructure long-running tasks t into a sequence of short-running tasks $t_1, \ldots, t_i$, other tasks executing between some t_j and t_{j+1} may interfere with t's intended atomic computation. Furthermore, tasks executing across several task-buffers (e.g., on a multi-core processor) are not guaranteed to execute serially, and may interleave their accesses to shared memory. Formal reasoning about concurrent program behavior is thus still crucial to prevent costly programming errors.

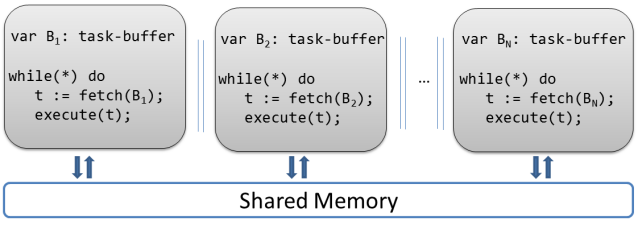

Figure 1. A general architecture of asynchronous programs with N task buffers. Tasks from the same buffer execute serially, but concurrently with tasks of other buffers.

In order to analyse asynchronous programs, we propose a formal model to capture concurrency in real-world asynchronous systems. In our model, each task has an associated *priority level*, and tasks execute from and post to *multiple* task-buffers. Tasks from each buffer execute serially, one after the other, posting new tasks to the same buffer from which they were taken, but tasks of distinct buffers may interleave. (Initially, each task-buffer contains at least one initial task.) When one task completes, a highest-priority pending task is taken from its buffer and begins executing. At any moment when a task t_1 posts a higher-priority task t_2, t_1 is suspended to execute t_2. When there are no more remaining tasks with higher priority, t_1 resumes execution. The number of priority levels and task-buffers are both finite and statically determined. Furthermore, tasks taken from one task-buffer may not post tasks to another; the only inter-buffer communication occurs though shared-memory.

Our model extends previous formal model of asynchronous programs proposed by Sen and Viswanathan [26] to more-accurately capture the concurrency in real-world asynchronous programs [2, 20, 21, 27]. Without accounting for priorities, the previous model permits priority-order breaking executions which can never arise in the actual system being modeled. By considering a single task-buffer, the previous model excludes inter-buffer task interleavings which can arise in actual systems. In the context of formal verification, the former leads to falsely detected errors, while the latter leads to uncaught errors.

We also give an algorithm for analyzing programs that follow our model. The state-reachability problem for finite-data programs with a single task-buffer and single priority level is decidable [9, 26]. It remains decidable in the presence of multiple levels, though highly complex [3]. Extending orthogonally to multiple interleaving task-buffers renders the problem undecidable: recursive multi-threaded programs are easily captured. Nevertheless, we believe the multiple task-buffer asynchronous model is important enough to be distinguished from the multi-threaded model for two reasons. First, encoding an asynchronous program with multiple prioritized task-buffers as a multi-threaded program requires adding additional state and synchronization to ensure (a) same-buffer tasks do not interleave, and (b) only highest-priority tasks from each buffer may execute. Encoding these constraints with general-purpose synchronization mechanisms disregards the more declarative program structure, and leads to inefficient program exploration (as shown by our experiments). Second, by leveraging the intensional structure of concurrency in the actual program, we can derive useful heuristics for prioritized program exploration. For example, following the premise of context-bounding [22, 24], we benefit by exploring program executions with relatively few alternations between task-buffers, or relatively few intra-buffer task reorderings (without directly restricting the number of tasks executed).

In the spirit of exploiting the structure of asynchronous programs, we develop a parameterized program analysis by reduction to sequential program analysis. The analysis parameters K_1 and K_2 restrict the amount of interleaving between tasks of different buffers (K_1) and reordering between tasks of the same buffer (K_2); as we increase K_1 (resp., K_2) our analysis explores more and more inter-buffer task interleavings (resp., intra-buffer task reoderings); in the limit as both K_1 and K_2 approach infinity, our encoding encodes all valid executions. For any given parameter values, we succinctly encode a limited set of asynchronous executions as executions of a non-deterministic sequential program with a polynomial number (in K_1 and K_2) of additional copies of shared variables. Our encoding is compositional in the spirit of existing sequential program reductions [4, 7, 13, 17], in that each task-buffer's execution is explored in isolation from other task-buffers, and the task-buffers themselves are not explicitly represented. Such compositionality sidesteps the combinatorial explosion that would arise by keeping the local-state of other buffers' tasks, and by explicitly representing the contents of even a single task-buffer.

Our reduction happens in two steps. First, in Section 4, we reduce asynchronous programs with multiple task-buffers to asynchronous programs with a single task-buffer, while preserving task priorities. We accomplish this by a code-to-code translation that introduces K_1 copies of shared variables. Each copy stores the value at which a task resumes after being preempted by other buffers' tasks; these values are initially guessed, and later validated. Interleavings with other task-buffers are thus simulated locally by moving to the next shared variable copy. In the second step (Section 5), we reduce single-buffer asynchronous programs with task priorities to sequential programs. We again accomplish this by a code-to-code translation, though in this translation we also introduce copies of shared variables for each priority level. Since our translation targets a sequential program without explicitly representing the task-buffer, each asynchronous task post is roughly translated as a synchronous procedure call. As posts to lower-priority tasks are not allowed to execute immediately, we use the extra shared variable copies to summarize their execution, postponing their effects until later.

This paper makes the following contributions:

- Motivated by our investigation into the concurrency arising in real-world desktop, server, and embedded asynchronous software (Section 2), we introduce a model of asynchronous programs with multiple task-buffers and task priorities (Section 3) that can naturally express the concurrency in these applications.
- We propose an incremental parameterized analysis technique for asynchronous programs by a two-step reduction to sequential programs (Sections 4 and 5).
- We demonstrate that our analysis is relatively easy to implement and efficient in practice. We have written a prototype implementation for bounded verification of asynchronous C programs (Section 6). Our tool is able to discover errors in our case studies, without imprecisely reporting false errors which would be detected by analyses based on existing asynchrony models.

By translating asynchronous programs to sequential programs, we allow the many existing sequential-program analysis tools to be lifted for (underapproximate) analysis of asynchronous programs. Moreover, our translation is agnostic to datatypes present in the program, and is thus able to target analyses that support arbitrary data-domains, e.g., Boolean programs, programs with integers, or lists, etc.

2. Asynchronous Programming in Practice

In order to build practical verification and debugging tools, we desire to formally specify the program behaviors that occur in reactive software systems. To better understand why existing formal programming models are inadequate, we examine two real-world applications: hardware-software interaction in the Windows operating system, and asynchronous multi-processing in the Apache web server.

2.1 Hardware-Software Interaction in the Windows Kernel

The primary mechanism for ensuring high-performance hardware interaction in the Windows kernel is *priority interrupt levels*. In the following discourse, we focus on three levels in decreasing priority order—`DEVICE_LEVEL`[1], `DISPATCH_LEVEL`, and `PASSIVE_LEVEL`.

At the `DEVICE_LEVEL` run the software "interrupt service routines" (ISRs). A Boolean-valued "interrupt line" connecting the device to a processor core triggers a fixed ISR: when a core's interrupt line is raised, and an ISR is not currently running, the currently-running code is interrupted to execute the ISR. Since `DEVICE_LEVEL` routines prevent any other code, including the scheduler, from executing, `DEVICE_LEVEL` routines should execute in a very short period of time, delegating remaining computation to an asynchronous "deferred procedure call" (DPC). The Windows kernel maintains a queue of pending DPCs and periodic invocations of the Windows scheduler, and executes each one-by-one at `DISPATCH_LEVEL` until completion, until the queue is empty. Normal applications run at `PASSIVE_LEVEL`, and thus, only execute when the DPC queue is empty. Like `DEVICE_LEVEL` code, DPCs should not sleep or block waiting for I/O; instead they should either continue to defer future work by queueing another DPC, or delegate the work to a `PASSIVE_LEVEL` thread. Although DPCs are guaranteed not to interleave with other DPCs nor application threads on the same core, DPCs may execute concurrently with ISRs, DPCs, the Windows scheduler, and threads, of other cores.

Besides bringing reactivity, the priority-level scheme provides synchronization for devices' shared data. Since code above `PASSIVE_LEVEL` executes atomically without preemption by same- or lower-level code, raising from `PASSIVE_LEVEL` to `DISPATCH_LEVEL` synchronizes device accesses on a single-core.

Our model can precisely capture these aspects of Windows by assigning each task one of the three priority levels, and by dividing code from separate cores into separate task-buffers. In order to capture arbitrary interleaved interaction between hardware and software, we can designate a separate task-buffer for a single spinning hardware-simulating task. Note that ignoring priorities can result in false data-race errors on device-data protected with level-based synchronization, and ignoring multiple buffers will miss real errors due to interleaving across separate cores (or with the hardware).

2.2 Multi-Processing in Apache via Grand Central Dispatch

In a recently-released software patch, the once multi-threaded Apache web server has been redesigned as an asynchronous program using the `libdispatch` concurrency framework [1]. In the updated architecture, the application begins by creating a number of concurrently-executing connection-listener objects, each maintaining a separate queue of incoming connection requests. Each listener handles connection requests by creating a new connection object, which besides storing all data relevant to a given client connection, maintains a queue of tasks pertaining to the connection. Client activity on the low-level network socket triggers additional connection-processing tasks to be placed on the queue. Tasks spurred by periodic timers and server replies are also placed on the queue. Importantly, the tasks manipulating the data of any given connection are placed in the same task-queue, although the connection-listening

[1] `DEVICE_LEVEL` is really a family of levels to which each device maps. Here we consider a generic device mapped to a single level.

tasks responsible for initializing new connection data are distributed across several queues.

The underlying concurrency manager, called Grand Central Dispatch (GCD), is responsible for executing tasks from the various queues. GCD ensures that each task from a queue executes only after the previous task has completed.[2] Concerning Apache, this ensures that at most one task per connection executes at any moment, and allows several tasks from distinct connections and connection listeners to execute concurrently.

For any finite number of connections and listeners, our programming model accurately captures the possible executions, by associating a task-buffer to each connection and connection listener. Existing formal models do not suffice. Single-buffer asynchronous programs could not capture potentially unsafe interleaving between connections and between connection-listeners; without adding extra synchronization, multi-threading allow tasks of the same connection to erroneously interleave their shared-memory accesses.

2.3 Abstraction of Task-Buffer Order

These systems fit well into our programing model with one caveat—we abstract the FiFo-ordered queues by unordered buffers. We believe this is justified for two reasons. First, algorithmic formal reasoning about processes accessing unbounded ordered queues remains a difficult problem.[3] Second, our understanding of these particular systems leads us to believe that while the FiFo semantics is important to ensures fairness and reactivity, key safety properties may not rely on the order. Of course, this is not a technical limitation—one can encode the FiFo order using shared-memory synchronization if required (but at the cost of introducing significant complexity).

3. Asynchronous Programs

We consider a programming model in which computations are divided into *tasks*. Each task has a fixed *priority-level* and a *task-buffer* associated with it, and behaves essentially as a recursive sequential program. Besides accessing a global memory shared by all other tasks, each task can *post* additional tasks to its task-buffer for later execution. Same-level tasks from each buffer execute serially, one after the other, yet are interrupted by higher-level tasks, and executed in parallel with tasks from distinct buffers. We call a task-buffer without a currently-executing task *idle*, the yet-to-be executed tasks of a buffer *pending*, the tasks who have finished execution *completed*, and the task chosen to execute when another completes *dispatched*. Initially each task-buffer is idle and contains at least one pending task. When a buffer is idle or a task completes, a highest-priority pending task is dispatched. At any moment when a task t_1 posts a higher-priority task t_2, t_1 is suspended to execute t_2—we say t_1 is *interrupted* by t_2; as soon as all higher-priority tasks of the same buffer have completed, t_1 resumes execution.

The resulting model generalizes the classical model of asynchronous programs [26] by adding priority-levels [3] and multiple task-buffers. Here, only highest-priority tasks may be dispatched, and tasks from different task buffers execute in parallel. Though we prefer to keep the concept of task-buffer disconnected from physical entities such as processes, threads, processors, cores, etc., Section 2 describes particular mappings to task-buffers from such entities in real-world asynchronous systems.

We adopt an *interleaving semantics* for our model: at any point in time only tasks from a single task-buffer may execute, but a *(buffer) preemption* can transfer control to a task from another buffer.

[2] Strictly speaking, GCD schedules at most w tasks from each queue, where w is a user-specified per-queue parameter, usually set to 1.

[3] State-reachability for even a single finite-state process accessing an unbounded queue is undecidable [5].

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

Figure 2. The grammar of asynchronous programs P. Here T is an unspecified type, $p \in \mathsf{Procs}$ ranges over procedure names, $e \in \mathsf{Exprs}$ over expressions, and $m \in M$ over priority levels.

We explicitly mark opportunities for such control transfers with a special program statement called **zield**. Additionally, to further extend the applicability of our model we introduce a similar control-transfer called a *(task) preemption* that suspends execution of the current task and transfers control to another same-level task of the same buffer. We mark opportunities for these transfers with a special program statement called **yield**. Having explicit instructions for such control transfer allows for great flexibility in model. We write atomic methods (e.g., synchronization operations) simply by omitting **yield**s and **zield**s for the duration of the atomic section. We model `DISPATCH_LEVEL` deferred procedure calls in Windows (see Section 2) by omitting **yield**s (but not **zield**s). Moreover, we model multithreaded programs by inserting **yield**s before every statement that accesses shared memory.[4]

3.1 Program Syntax

Let Procs be a set of procedure names, Vals a set of values, Exprs a set of expressions, and $N, M \in \mathbb{N}$, resp., the number of task-buffers and priority-levels. For $N \in \mathbb{N}$ we denote the set $\{0, \ldots, N-1\}$ simply by N. The grammar of Figure 2 describes our language of *asynchronous programs (with prioritized task-buffers)*. We intentionally leave the syntax of expressions e unspecified, though we do insist Vals contains `true` and `false`, and Exprs contains Vals and the *(nullary) choice operator* $\star$.

A *multi-buffer* (resp., *single-buffer*) *program* is a program with (resp., without) **zield** statements. A program in which the first statement of each posted procedure[5] is **while** $\star$ **do yield** is called *reorderable*, a program in which the **yield** statement does (resp., does not) occur otherwise (including transitively called procedures) is called *preemptive* (resp., *non-preemptive*), and a program without **yield** or **zield** statements is called *scheduler-dependent*. Intuitively, a scheduler-dependent program is deterministic modulo nondeterminism in the sequential statements (e.g., arising by using the $\star$ operator in **if-then-else** statements), and modulo the possibly nondeterministic choices made at task-dispatch points; in a reorderable program, any dispatched task can immediate re-become pending. A *sequential* program is one without **post**, **yield**, and **zield** statements.

Each program P declares a single shared type-T global variable `g`, and a sequence of procedures named $p_1 \ldots p_i \in \mathsf{Procs}^*$, each p having single type-T parameter `l` and a top-level statement denoted s_p. The set of program statements s is denoted Stmts. Furthermore each program P declares N parameter-less initial procedures named $\texttt{main}(0), \texttt{main}(1), \ldots, \texttt{main}(N-1)$, which are never posted nor called; we will assume an initial frame of $\texttt{main}(n)$ is initially pending for each task-buffer $n \in N$. Intuitively, a **post** $m\ p\ e$ statement is an asynchronous call to a procedure p with argument e to be executed at priority level m. The **yield** statement transfers control to a same-priority level pending task of the same task-buffer,

[4] Here we assume memory accesses are sequentially consistent [18].

[5] We assume each procedure can be either posted or called, but not both.

$$g, \ell, v \in \mathsf{Vals} \qquad f \in \mathsf{Frames} \stackrel{\text{def}}{=} \mathsf{Vals} \times \mathsf{Stmts} \times M$$
$$e \in \mathsf{Exprs} \qquad t \in \mathsf{Tasks} \stackrel{\text{def}}{=} \mathsf{Frames}^*$$
$$s \in \mathsf{Stmts} \qquad a \in N \to \mathsf{Tasks}$$
$$p \in \mathsf{Procs} \qquad b \in N \to \mathbb{M}[\mathsf{Tasks}]$$
$$c \in \mathsf{Configs} \stackrel{\text{def}}{=} \mathsf{Vals} \times N \times (N \to \mathsf{Tasks}) \times (N \to \mathbb{M}[\mathsf{Tasks}])$$

Figure 3. The semantic domains of the meta-variablesused in the transition relation of Figure 4.

and the **zield** statement transfers control to a task from another task-buffer. The **assume** e statement proceeds only when e evaluates to `true`; we will use this statement to block undesired executions in subsequent reductions to sequential programs.

The programming language we consider is simple, yet very expressive, since the syntax of expressions is left free, and we lose no generality by considering only single global and local variables. Appendix A lists several syntactic extensions, which easily reduce to the syntax of our grammar, and which we use in the source-to-source translations of the subsequent sections. Additionally, hardware interrupts and lock-based synchronization can be encoded using global shared memory; see Sections 3.3 and 3.4.

3.2 Program Semantics

A *(procedure-stack) frame* $f = \langle \ell, s, m \rangle$ is a current valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable `l`, along with a statement s to be executed, and a priority level $m \in M$. (Here s describes the entire body of a procedure p that remains to be executed, and is initially set to p's top-level statement s_p.) The set of frames is denoted $\mathsf{Frames} \stackrel{\text{def}}{=} \mathsf{Vals} \times \mathsf{Stmts} \times M$. A sequence $t = f_1 \dots f_i$ of frames is called a *task* (f_1 is the top-most frame), and the set of tasks is denoted Tasks. The *level* (resp., *base level*) of a task t, denoted by $\mathsf{level}(t)$ (resp., $\mathsf{base}(t)$), is the level of t's topmost (resp., bottommost) frame, and -1 when $t = \varepsilon$.

Since we consider a cooperative multitasking model, where control transfers between buffers and between tasks of the same buffer are explicit, our operational model need only consider a single currently executing task of a single buffer at any moment. When higher-level tasks of the same buffer interrupt a lower-level task, our model simply adds the higher-level task's frame to the top of the current procedure stack. We thus represent each task-buffer $n \in N$ by a currently-executing active task $a(n) \in \mathsf{Tasks}$, along with a multiset $b(n) \in \mathbb{M}[\mathsf{Tasks}]$ of pending tasks, and we define a *configuration* $c = \langle g, n, a, b \rangle$ as a valuation $g \in \mathsf{Vals}$ of the global variable `g`, along with a currently active task-buffer $n \in N$, an active task $a : N \to \mathsf{Tasks}$ per buffer, and a multiset of pending tasks $b : N \to \mathbb{M}[\mathsf{Tasks}]$ per buffer. Figure 3 summarizes the various semantic domains and meta-variable naming conventions.

We define $\mathsf{top} : \mathbb{M}[\mathsf{Tasks}] \to M$ as the maximum level for which there is a pending task in the given buffer:

$$\mathsf{top}(\mu) \stackrel{\text{def}}{=} \max(\{0\} \cup \{\mathsf{level}(t) : t \in \mu\}),$$

and we let $\mathsf{top}(\langle g, n, a, b \rangle) \stackrel{\text{def}}{=} \mathsf{top}(b(n))$ be the maximum such level for the currently active buffer. Similarly, we define the level of a configuration as the level of the currently active buffer: $\mathsf{level}(\langle g, n, a, b \rangle) \stackrel{\text{def}}{=} \mathsf{level}(a(n))$.

The singleton pending-tasks map $(n \mapsto t)$ maps n to $\{t\}$, and $n' \in N \setminus \{n\}$ to $\emptyset$, and the union $b_1 \cup b_2$ of pending-tasks maps is defined by the multiset union of each mapping: $(b_1 \cup b_2)(n) \stackrel{\text{def}}{=} b_1(n) \cup b_2(n)$ for $n \in N$. Similarly, the active-task map $a' = a(n \mapsto t)$ extends a by setting the currently active task of buffer n to t: $a'(n) = t$, and $a'(n') = a(n')$ for all $n' \in N \setminus \{n\}$.

For expressions without program variables, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ such that $[\![\star]\!]_{\mathrm{e}} = \mathsf{Vals}$. For convenience, when $c = \langle g, n, a, b \rangle$ and $a(n) = f \cdot t = \langle \ell, s, m \rangle\, t$, we define

$$e(c) \stackrel{\text{def}}{=} e(g, f \cdot t) \stackrel{\text{def}}{=} e(g, f) \stackrel{\text{def}}{=} e(g, \ell) \stackrel{\text{def}}{=} [\![e[g/\mathtt{g}, \ell/\mathtt{l}]]\!]_{\mathrm{e}}$$

to evaluate the expression e in a configuration c (alternatively, in a global valuation g and task $f \cdot t$) by substituting the current values for variables `g` and `l`. As these are the only program variables, the substituted expression $e[g/\mathtt{g}, \dots]$ has no free variables. For expressions e over only the task-local variable `l` we write $e(f \cdot t) \stackrel{\text{def}}{=} [\![e[\ell/\mathtt{l}]]\!]_{\mathrm{e}}$. Additionally we define

$$c(\mathtt{g} \leftarrow g') \stackrel{\text{def}}{=} \langle g', n, a, b \rangle \qquad \text{global assignment}$$
$$c(\mathtt{l} \leftarrow \ell') \stackrel{\text{def}}{=} \langle g, n, a(n \mapsto f' \cdot t), b \rangle \qquad \text{local assignment}$$
$$c \cdot f_0 \stackrel{\text{def}}{=} \langle g, n, a(n \mapsto f_0 \cdot f \cdot t), b \rangle \qquad \text{frame push/pop}$$

where $f' = \langle \ell', s, m \rangle$ updates the local valuation of frame f, and f_0 is an arbitrary frame.

To reduce clutter and highlight the meaningful components of rules in the operational program semantics, we introduce a notion of context. A *statement context* S is a term derived from the grammar $S ::= \diamond \mid S;\ s$, where $s \in \mathsf{Stmts}$. We write $S[s]$ for the statement obtained by substituting a statement s for the unique occurrence of $\diamond$ in S. A *task-statement context* $T = \langle \ell, S, m \rangle\, t$ is a task whose top-most frame's statement is replaced with a statement context, and we write $T[s]$ to denote the task $\langle \ell, S[s], m \rangle\, t$. Finally, a *configuration-statement context* $C = \langle g, n, a(n \mapsto T), b \rangle$ is a configuration whose currently active task is replaced with a task-statement context, and we write $C[s]$ to denote the configuration $\langle g, n, a(n \mapsto T[s]), b \rangle$. Intuitively, a context filled with s, e.g., $C[s]$, indicates that s is the next statement to execute in a given configuration, task, or statement sequence. We abbreviate $e(T[\mathbf{skip}])$, $e(g, T[\mathbf{skip}])$, and $e(C[\mathbf{skip}])$ by, resp., $e(T)$, $e(g, T)$, and $e(C)$ for arbitrary expressions e. As an example, the ASSUME rule of Figure 4 takes a configuration $C[\mathbf{assume}\ e] = \langle g, n, a(n \mapsto \langle \ell, \mathbf{assume}\ e;\ s, m \rangle\, t), b \rangle$ in which $\mathtt{true} \in e(C) = e(g, \ell)$ and transitions to the configuration $C[\mathbf{skip}] = \langle g, n, a(n \mapsto \langle \ell, \mathbf{skip};\ s, m \rangle\, t), b \rangle$; as only the current statement is updated, the description $C[\mathbf{assume}\ e] \to_P C[\mathbf{skip}]$ completely and concisely describes the configuration change.

Figure 4 defines the transition relation $\to$ of asynchronous programs as a set of operational steps on configurations. The transitions for the sequential statements are mostly standard. 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 reductions between program models in the subsequent sections.)

More interesting are the transitions for the asynchronous constructs (i.e., **post**, **yield**, and **zield**). The POST rule creates a new frame to execute given procedure p with argument e at level m', and places the new frame in the pending-tasks of the currently active task-buffer n. When a (single-frame) task f completes its execution, the RESUME rule discards f to continue the execution of the task t below f on the procedure stack. The YIELD rule allows a task to relinquish control to another pending task at the same level. The ZIELD rule simply updates the currently active task-buffer from n to some $n' \in N$. The DISPATCH rule schedules a highest-level task t_1 when the currently executing task t_2 has a lower level.

An *execution of a program P (from c_0 to c_j)* is a configuration sequence $c_0 c_1 \dots c_j$ such that $c_i \to_P c_{i+1}$ for $0 \le i < j$. A configuration $c = \langle g, n_0, a, b \rangle$ of a program P is *g_0-initial* when $g = g_0$, $n_0 = 0$, and for all $n \in N$,

$$a(n) = \varepsilon \qquad \text{and} \qquad b(n) = \{\langle \ell, s_{\mathtt{main}(n)}, 0 \rangle\}$$

$$\textsc{Skip}\ \frac{}{C[\mathbf{skip};\ s] \underset{P}{\rightarrow} C[s]} \qquad \textsc{Assume}\ \frac{\texttt{true} \in e(C)}{C[\mathbf{assume}\ e] \underset{P}{\rightarrow} C[\mathbf{skip}]} \qquad \textsc{If-Then}\ \frac{\texttt{true} \in e(C)}{C[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \underset{P}{\rightarrow} C[s_1]} \qquad \textsc{If-Else}\ \frac{\texttt{false} \in e(C)}{C[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \underset{P}{\rightarrow} C[s_2]}$$

$$\textsc{Assign}\ \frac{v \in e(C)}{C[x := e] \underset{P}{\rightarrow} C[\mathbf{skip}]\ (x \leftarrow v)} \qquad \textsc{Loop-Do}\ \frac{\texttt{true} \in e(C)}{C[\mathbf{while}\ e\ \mathbf{do}\ s] \underset{P}{\rightarrow} C[s; \mathbf{while}\ e\ \mathbf{do}\ s]} \qquad \textsc{Loop-End}\ \frac{\texttt{false} \in e(C)}{C[\mathbf{while}\ e\ \mathbf{do}\ s] \underset{P}{\rightarrow} C[\mathbf{skip}]}$$

$$\textsc{Call}\ \frac{v \in e(C) \qquad m = \mathsf{level}(C)}{C[\mathbf{call}\ x := p\ e] \underset{P}{\rightarrow} C[x := \star] \cdot \langle v, s_p, m\rangle} \qquad \textsc{Return}\ \frac{g \in \mathsf{g}(C) \qquad v \in e(g, \ell) \qquad m = \mathsf{level}(C)}{C[x := \star] \cdot \langle \ell, S[\mathbf{return}\ e], m\rangle \underset{P}{\rightarrow} C[x := v]} \qquad \textsc{Resume}\ \frac{m > \mathsf{level}(c)}{c \cdot \langle \ell, S[\mathbf{return}\ e], m\rangle \underset{P}{\rightarrow} c}$$

$$\textsc{Post}\ \frac{\ell \in e(g, T) \qquad f = \langle \ell, s_p, m\rangle \qquad b' = b \cup (n \mapsto f)}{\langle g, n, a(n \mapsto T[\mathbf{post}\ m\ p\ e]), b\rangle \underset{P}{\rightarrow} \langle g, n, a(n \mapsto T[\mathbf{skip}]), b'\rangle} \qquad \textsc{Dispatch}\ \frac{\mathsf{level}(t_1) \geq \mathsf{top}(b(n)) > \mathsf{level}(t_2)}{\langle g, n, a(n \mapsto t_2), b \cup (n \mapsto t_1)\rangle \underset{P}{\rightarrow} \langle g, n, a(n \mapsto t_1 \cdot t_2), b\rangle}$$

$$\textsc{Yield}\ \frac{\mathsf{level}(T_1) = \mathsf{base}(T_1) > \mathsf{level}(t_2) \qquad b' = b \cup (n \mapsto T_1[\mathbf{skip}])}{\langle g, n, a(n \mapsto T_1[\mathbf{yield}] \cdot t_2), b\rangle \underset{P}{\rightarrow} \langle g, n, a(n \mapsto t_2), b'\rangle} \qquad \textsc{Zield}\ \frac{n_2 \in N}{\langle g, n_1, a(n_1 \mapsto T[\mathbf{zield}]), b\rangle \underset{P}{\rightarrow} \langle g, n_2, a(n_1 \mapsto T[\mathbf{skip}]), b\rangle}$$

Figure 4. The transition relation for the asynchronous programs; each rule, besides DISPATCH, is implicitly guarded by $\mathsf{level}(c) \geq \mathsf{top}(c)$, where c is the configuration on the left-hand side of the transition.

for some/any[6] $\ell \in \mathsf{Vals}$. A configuration $\langle g, n, a, b\rangle$ is *g_f-final* when $g = g_f$. A pair $\langle g_0, g_f\rangle$ is a *reachability fact of P* when there exists an execution of P from some c_0 to some c_f such that c_0 is g_0-initial and c_f is g_f-final.

Problem 1 (State-Reachability)**.** *The* state-reachability problem *is to determine, given a pair $\langle g_0, g_f\rangle$ and a program P, whether $\langle g_0, g_f\rangle$ is a reachability fact of P.*

Since preemption (i.e., arbitrary use of the **yield** statement) and multiple task-buffers can both be used to simulate arbitrary multi-threaded programs, the presence of either feature makes state-reachability undecidable.

Theorem 1. *The state-reachability problem is undecidable for finite-data asynchronous programs with either preemptive tasks, or with multiple task-buffers.*

For single-buffer programs without arbitrary preemption between tasks, Atig et al. [3] have shown decidability by reduction to reachability in a decidable class of Petri nets with inhibitor arcs.

Theorem 2. *The state-reachability problem is decidable for finite-data single-buffer programs without preemption.*

Though Atig et al. [3]'s reduction does show decidability, it does not lead to a practical program analysis algorithm since the only known algorithms for checking reachability in Petri nets have extremely high complexity—they are non-primitive recursive [8]. In the following sections we design an approximating algorithm for the general case, encompassing both cases of Theorem 1 and 2, by reduction to sequential program analysis. Though approximation is necessary in the undecidable case with preemption or multiple buffers, approximation also allows us practical analysis algorithms in the decidable yet complex case with prioritized buffers. Note that though these decidability results only apply to finite-data programs, our sequential reduction applies equally well to programs with infinite data, and does not approximate individual program states; e.g., given a program using unbounded integer variables, our translation encodes a subset of all possible concurrent behaviors, without abstracting in any given state the integer variable valuations.

[6] Since $\texttt{main}(n)$ takes no arguments, the initial local valuation is irrelevant.

3.3 Modeling Hardware Interrupts

Although our model only allows tasks to post other tasks to their own buffers, inter-buffer task-posting can be modeled using shared memory. For instance, consider modeling hardware interrupts in operating systems kernels like Windows and Linux. Each processor core has a fixed number of Boolean-valued interrupt lines which once raised by hardware devices, trigger software routines to interrupt the currently-running application or kernel task (see Section 2.1 for more details).

To model hardware interrupts we add an additional highest priority-level M, and an array $\mathbf{var}\ \texttt{IRQ}[N]\texttt{:}\ \mathbb{B}$ to the global program state—here we suppose each core corresponds to a task-buffer, and we handle only a single interrupt line per core; the generalization to multiple lines per core is straightforward. For each core $n \in N$, we designate a fixed procedure $\mathbf{proc}\ \texttt{ih}(n)\ \texttt{()}\ s$ to be the interrupt handler for the core. Raising an interrupt on core $n \in N$ is as simple as setting $\texttt{IRQ}[n]\ \texttt{:=}\ \texttt{true}$. An interrupt is caught and processed on core n by inserting the following code before each global-variable access and **post**, **yield**, or **zield** statement

```
if IRQ[n] then
    IRQ[n] := false;
    post M ih(n) ()
```

Posting to level M ensures that no matter which level the current task is running at, the interrupt handler will interrupt it.

3.4 Modeling Synchronization

As the execution of tasks across multiple task-buffers may interleave, in general programs need a way to ensure atomicity of data accesses. Implementing atomic locking instructions is not difficult because we have assumed tasks are only preempted by other buffer's tasks at designated **zield** points, or by same-buffer tasks at designated **yield** points. A lock can thus be implemented by adding an additional global variable $\mathbf{var}\ \texttt{lock}\texttt{:}\ \mathbb{B}$; acquiring the lock is achieved by the statement

```
while lock = true do
    yield; // only for preemptive programs
    zield;
lock := true
```

and releasing the lock is as simple as setting `lock := false`. Note that the exit from the **while** loop and the assignment `lock := true` are guaranteed to happen atomically, since there are no interfering **yield** or **zield** statements. Once the lock is held by one task, any other acquiring tasks must wait until it is released.

4. Reduction to Single-Buffer Programs

As a first step in our scheme to reduce the state-reachability problem of asynchronous programs to state-reachability in sequential programs, we translate multiple-buffer programs to single-buffer programs, while preserving task priorities. As the state-reachability problem for single-buffer finite-data asynchronous programs with task priorities is decidable [3] for non-preemptive programs[7], while our multi-buffer variation is not, our translation necessarily represents an approximation of the original problem. Though our translation encodes various inter-buffer interleavings, it cannot encode every inter-buffer interleaving. In order to control and refine the amount of considered interleavings, and thus the degree of approximation, our translation takes a bounding parameter K. Following the approach of the original multi-threaded sequentializations [17], for a given bounding parameter K, we explore only K-round round-robin executions in buffer-index order; i.e., in each round, tasks from buffer 0 execute until a (perhaps not the first) **zield** statement, at which point tasks from buffer 1 execute to some **zield** statement, and so on. At the **zield** statement where a task from buffer $N-1$ gives up control, the second round begins, resuming the suspended task of buffer 0. Note that the bounding permits arbitrarily long executions within a buffer.

Example 1 (Restricted Inter-Buffer Interleaving)**.** The following asynchronous program with 2 task buffers and a single priority demonstrates how K-round exploration restricts program behaviors.

```
var b: 𝔹                    proc p ()
var r: ℕ                        zield;
                                assume !b;
proc main(0) ()                 b := true;
    b := true;                  r := r + 1;
    r := 1;                     post 0 p ();
    post 0 p;                   return
    return
                            proc q ()
proc main(1) ()                 zield;
    post 0 q                    b := false;
    return                      post 0 q ();
                                return
```

In the round-robin order, execution begins with `main`(0) of the first task-buffer, which sets the variable `b` to `true`, sets `r` to `1`, and posts a single task `p`. Each `p` task blocks unless `b` is set to `false`, in which case `b` is set to `true`, `r` incremented, and `p` re-posted. When the **zield** statement is encountered, control may be transferred to the second task buffer, which begins in `main`(1) by posting `q`. Each instance of `q` sets `b` to `false`, and re-posts `q`.

In a single-round execution, i.e., $K = 1$, the only reachable value of `r` is `1`. In order to increment `r`, a `q` task of the second buffer, which sets `b` to `false`, must be executed before `p`'s **assume** statement. In general, incrementing `r` K times requires K rounds of execution, each in which a `q` task from the second buffer proceeds a `p` task of the first. Since only one such alternation can happen per round, K rounds are required to make K alternations. □

[7] Note that the programs Atig et al. [3] consider are non-preemptive; what they refer to as "preemption" we refer to as "interruption."

```
// translation                  // translation
// of var g: T                  // of post m p e
var G[K]: T                     post (m+1) p e
var k: K
                                // translation of
// translation of g             // proc main(i) () s
G[k]                            proc main'(i) () s
                                proc main(0) ()
// translation                     for n = 0 to N − 1 do
// of zield                        k := 0;
k := in(k,K-1)                      post 1 main'(n) ()
```

Figure 5. The multi-to-single buffer code translation $(\!(P)\!)^K_{\mathrm{M}}$ of a multi-buffer asynchronous program P parameterized by $K \in \mathbb{N}$. The resulting single-buffer program encodes only round-robin task-buffer executions of K rounds.

As in Lal and Reps [17]'s original multi-threaded sequentialization, our code translation $(\!(\cdot)\!)^K_{\mathrm{M}}$ of Figure 5 stores a copy of the global valuation reached in each round. Initially, the global valuations for the second round and beyond are set non-deterministically. Then, for each task buffer n, we execute all tasks from n across all rounds before moving on to the next buffer and resetting the round to 0. At non-deterministically chosen **zield** statements, any task may cease accessing the i^{th} global valuation copy and begin using the $(i+1)^{\text{st}}$ copy; this simulates the progression of buffer n from round i to round $(i+1)$. After each buffer has completed executing all of its tasks, we can determine whether the initially guessed global valuations were valid by ensuring the initial valuation from each round $i > 0$ matches the valuation reached in round $(i-1)$. This validity relation is captured formally by a predicate S^K_{M} relating initial and final global valuations: a "sequentialized" execution of $(\!(P)\!)^K_{\mathrm{M}}$ from g_0 to g_f represents a valid K-round round-robin execution of the original program P only when $S^K_{\mathrm{M}}(g_0, g_f)$, in which case a mapping f^K_{M} translates the reachability pair $\langle g_0, g_f\rangle$ to a valid reachability pair $f^K_{\mathrm{M}}(g_0, g_f)$ in P. Note that our simulation requires only a single task-buffer since we execute all tasks of each buffer to completion before moving on to the next. To ensure all tasks of each buffer complete before coming back to `main`'s loop, we shift all priority levels up by one.

Formally, our *K-round multi-to-single buffer translation*

$$\Theta^K_{\mathrm{M}} = \left\langle (\!(\cdot)\!)^K_{\mathrm{M}}, S^K_{\mathrm{M}}, f^K_{\mathrm{M}} \right\rangle$$

is the code transformation $(\!(\cdot)\!)^K_{\mathrm{M}}$ listed in Figure 5, along with a validity predicate $S^K_{\mathrm{M}} : \mathsf{Vals}^2 \to \mathbb{B}$, indicating when an execution of $(\!(P)\!)^K_{\mathrm{M}}$ corresponds to a valid execution of P, and a function $f^K_{\mathrm{M}} : \mathsf{Vals}^2 \to \mathsf{Vals}^2$, mapping reachability pairs of $(\!(P)\!)^K_{\mathrm{M}}$ to reachability pairs of P:

- $S^K_{\mathrm{M}}(g_0, g_f) \stackrel{\text{def}}{=} \texttt{G[1..K-1]}(g_0) = \texttt{G[0..K-2]}(g_f)$,
- $f^K_{\mathrm{M}}(g_0, g_f) \stackrel{\text{def}}{=} \langle \texttt{G[0]}(g_0), \texttt{G[K-1]}(g_f)\rangle$.

Given this correspondence, we reduce state-reachability of K-round multi-buffer round-robin executions of P to state-reachability of $(\!(P)\!)^K_{\mathrm{M}}$; this reduction thus under-approximates P's behavior.

Theorem 3 (Soundness)**.** *For all programs P, if $\langle g_0, g_f\rangle$ is a reachability fact of $(\!(P)\!)^K_{\mathrm{M}}$ and $S_M(g_0, g_f)$ holds, then $f_M(g_0, g_f)$ is a reachability fact of P.*

Since every multi-buffer execution can be expressed in a finite number of rounds of a round-robin execution, our reduction completely captures all executions in the limit as K approaches infinity.

Theorem 4 (Completeness)**.** *For all reachability facts $\langle g_0, g_f \rangle$ of a program P there exists $K \in \mathbb{N}$ and a reachability fact $\langle g'_0, g'_f \rangle$ of $(\!(P)\!)^K_M$ such that $\langle g_0, g_f \rangle = f_M(g'_0, g'_f)$ and $S_M(g'_0, g'_f)$.*

4.1 Translation Composition

Crucial to our staged-translation approach is the fact that a sequence of translations can be composed. The ordered composition of the translations $\Theta_1 = \langle (\!(\cdot)\!)_1, S_1, f_1 \rangle$ and $\Theta_2 = \langle (\!(\cdot)\!)_2, S_2, f_2 \rangle$ is defined by the translation $\Theta_2 \circ \Theta_1 = \langle (\!(\cdot)\!), S, f \rangle$ such that

- $(\!(\cdot)\!) = (\!(\cdot)\!)_2 \circ (\!(\cdot)\!)_1$,
- $S = S_2 \wedge (S_1 \circ f_2)$, and
- $f = f_1 \circ f_2$.

When both Θ_1 and Θ_2 are sound and complete in the sense of Theorems 3 and 4, then $\Theta_2 \circ \Theta_1$ is also sound and complete.

5. Reduction to Sequential Programs

In the second step of our reduction scheme, we translate single-buffer asynchronous programs with task priorities to sequential programs. Though the state-reachability problem is decidable for finite-data single-buffer programs without preemption [3], allowing arbitrary inter-task preemption does make the state-reachability problem undecidable, due to unbridled interleaving between concurrently executing tasks of the same level; indeed recursive multi-threaded programs are easily encoded when use of the yield statement is unrestricted. Furthermore, even in the decidable case of non-preemptive programs, the complexity of state-reachability is very high: due to the arbitrary dispatch order of same-level tasks (and perhaps further complicated by interruptions from higher-level tasks), the problem is at least as hard as reachability in Petri nets [3]—a problem for which the only known algorithms are non-primitive recursive [8].

Thus, as in Section 4, our translation to sequential programs is again obliged (in the presence of preemption), or at least better off (even without preemption), to represent only an approximation of the original state-reachability problem. Our translation encodes only a subset of the possible task interleavings and dispatch orderings. As before, we introduce a bounding parameter K with which to restrict the amount of interleavings expressed; with increasing values of K we capture more and more interleaving and task dispatching orders, and in the limit encode all possible executions.

To simplify our translation, we again divide our work in half. In Section 5.1, we propose a translation to remove yields from a given program with task priorities. In Section 5.2, we then translate a yield-free single-buffer asynchronous program with task priorities to a sequential program. The first "yield-elimination" step is crucial to our translation; it deals with both task reorderings (where any of the pending tasks at the same level can be dispatched) and task preemption. Our second step then simulates only a *fixed* schedule of task dispatching, following the approach of delay-bounded scheduling [7]. To ensure our translation encodes all possible task dispatch schedules in the limit as the bounding parameter approaches infinity, we assume programs are reorderable, i.e., by requiring that the first statement of each task is **while** $\star$ **do yield**. (Note that this does not make a program preemptive; recall the definitions of Section 3.1.)

5.1 Eliminating Preemption and Reordering

Again following the vectorization approach pioneered by Lal and Reps [17], we proceed by restricting the set of interleaved executions

```
// translation of
// call x := p e
call (x,k) :=
    p (e,cur_lvl,k)

// translation
// of return e
return (e,k)

// translation
// of var g: T
var G[K]: T
var R[M]: K

// translation
// of yield
k := in(k..K-1)

// translation of g
G[k]
```

```
// translation of
// proc p (var l: T) s
proc p (var l: T, cur_lvl: M,
          k: K) s

// translation of post m p e
2 assert m ≤ cur_lvl+1;
  if m = cur_lvl+1 then
4   let saved_G = G in
    let G? in
6   G := G?;
    G[0] := saved_G[k];
8   R[cur_lvl] := k;
    post m p (e,m,0);
10  assume G?[1..K-1] = G[0..K-2];
    saved_G[k] := G[K-1];
12  G := saved_G;
  else if m = cur_lvl then
14  post m p (e,m,k)
  else post m p (e,m,R[m])
```

Figure 6. The yield-eliminating translation $(\!(P)\!)^K_Y$ of a single-buffer multi-level program P, parameterized by $K \in \mathbb{N}$.

between tasks of any given level to those according to a round-robin schedule; for a given bounding parameter K, we will explore executions in which each task can be preempted by or reordered with other same-level tasks across K rounds.

To accomplish this, our code translation $(\!(\cdot)\!)^K_Y$ of Figure 6 stores a copy `G[`k`]` of the global valuation reached in each round k, for just one level at a time, and each task stores a current-round counter `k`. Initially execution begins with a task at level 0 accessing the 0th copy of the global valuation; the other copies of the global valuation are set non-deterministically. At any point upon encountering a **yield** statement, the currently executing task can increment his round counter to any value $k < K$, and begin accessing the k^{th} copy of the global valuation. Such an increment propels the current task to round k, simulating a preemption by or reordering with other tasks from his level. When a task in round k posts a same-level task, the posted task is constrained to execute in or after round k. The possibility of inter-level posts makes our reduction significantly more intricate than previous sequentializations [7, 17].

Posts to higher-level tasks interrupt execution of the currently executing task. When such a post happens, we save the global-valuation vector for the current level (Line 4), and allocate a new global-valuation vector for the target level whose first element is initialized with the current global valuation reached by the posting task (Lines 5–7); the other values are again guessed non-deterministically. When control returns to the posting task, we must ensure the guessed global valuations at the beginning of each round of the target level have been reached by previous rounds (Line 10); when this is the case we have simulated some round-robin execution of the target level's tasks. As long as those guesses can be validated, we restore the previously-saved global-valuation vector for the posting task's level, update the current-round's valuation with the final valuation reached in the posted task's level, and continue executing the posting task (Lines 11–12).

Though posting a lower-priority-level task is handled almost identically to posting a same-level task, there is one important difference: the round of the posted task t_2 must not occur before the round of an interrupted task t_1 of the same priority level waiting below on the task stack; otherwise causality is broken, since t_2 would execute before t_1 in the simulated execution, though t_2's existence may rely on t_1's execution. To prevent such anomalies, we store in `R[`m`]` the current round k of each priority level m below

the currently executing task, and constrain tasks posted to level m to execute no sooner than round k. To simplify our translation, we assume priority-level m tasks post only tasks of priority at most $m+1$; posts to higher levels can be encoded by a chain of posts.

We define formally the *K-round yield-eliminating translation*

$$\Theta_{\mathrm{Y}}^{K} = \left\langle (\!|\cdot|\!)_{\mathrm{Y}}^{K}, S_{\mathrm{Y}}^{K}, f_{\mathrm{Y}}^{K} \right\rangle$$

as the code transformation $(\!|\cdot|\!)_{\mathrm{Y}}^{K}$ listed in Figure 6, along with a validity predicate S_{Y}^{K} and reachability-fact map f_{Y}^{K}:

- $S_{\mathrm{Y}}^{K}(g_0, g_f) \stackrel{\text{def}}{=} \texttt{G[1..K-1]}(g_0) = \texttt{G[0..K-2]}(g_f)$,
- $f_{\mathrm{Y}}^{K}(g_0, g_f) \stackrel{\text{def}}{=} \langle \texttt{G[0]}(g_0), \texttt{G[K-1]}(g_f) \rangle$.

Given this correspondence, we reduce state-reachability of K-round round-robin intra-level executions of P to state-reachability of $(\!|P|\!)_{\mathrm{Y}}^{K}$; we thus under-approximate the behavior of P by restricting the set of interleavings/reorderings between same-level tasks.

Theorem 5 (Soundness)**.** *For all programs P, if $\langle g_0, g_f \rangle$ is a reachability fact of $(\!|P|\!)_{Y}^{K}$ and $S_{\mathrm{Y}}^{K}(g_0, g_f)$, then $f_{\mathrm{Y}}^{K}(g_0, g_f)$ is a reachability fact of P.*

Since every execution with yields can be expressed by allocating some finite number of rounds to each execution sequence of same-level tasks, our reduction completely captures all executions in the limit as K approaches infinity.

Theorem 6 (Completeness)**.** *For all reachability facts $\langle g_0, g_f \rangle$ of a program P there exists $K \in \mathbb{N}$ and a reachability fact $\langle g_0', g_f' \rangle$ of $(\!|P|\!)_{Y}^{K}$ such that $\langle g_0, g_f \rangle = f_{\mathrm{Y}}^{K}(g_0', g_f')$ and $S_{\mathrm{Y}}^{K}(g_0', g_f')$.*

5.2 Sequentializing Asynchronous Programs with Priorities

By removing all **yield** statements in the previous step, our final translation need only give a translation for scheduler-dependent single-buffer asynchronous programs with priorities. Our reduction expresses all executions according to a particular total order on task-dispatching. Though not strictly important for the soundness nor completeness of our encoding, the determined task-dispatch order roughly corresponds to Emmi et al. [7]'s deterministic depth-first schedule, except here we are faced with the added complexity of multiple priority levels.

Example 2 (Restricted Intra-Buffer Reordering)**.** The following reorderable single-buffer asynchronous program demonstrates how our K-bounded exploration restricts program behaviors. Execution begins with `main`, which sets the variable `b` to `true`, sets `r` to `1`, and posts a sequence of `p` tasks followed by a sequence of `q` tasks. Each `p` task blocks unless `b` is set to `false`, in which case `b` is set to `true` and `r` incremented. Each `q` task sets `b` to `false`.

```
var b: 𝔹                      proc p ()
var r: ℕ                          while ⋆ do yield;
                                  assume !b;
proc main ()                      b := true;
    b := true;                    r := r + 1;
    r := 1;                       return
    while ⋆ do
        post 0 p ();          proc q ()
    while ⋆ do                    while ⋆ do yield;
        post 0 q ();              b := false;
    return                        return
```

When $K = 1$, we explore only executions according to a deterministic schedule of task dispatching; specifically, according to the depth-first scheduler [7]. For this program, that means all posted `p` tasks must run before any `q` task, in which case the only reachable value of `r` is `1`. When $K > 1$, we introduce task reordering by allowing each task to advance among the K rounds at **yield** points. In this way, $K+1$ rounds suffices to capture K alternations from `q` to `p` tasks, allowing the value of `r` to be incremented K times. □

```
                                  // translation of
                                  // proc p (var l: T) s
                                  proc p (var l: T, cur_lvl: M) s
// translation of
// call x := p e
call x :=                         // translation of post m p e
    p (e,cur_lvl)                 2 assert m ≤ cur_lvl+1;
                                    if m = cur_lvl+1 then
// translation                    4     let g? = ⋆ in
// of var g: T                          G[m] := g?;
var done: 𝔹                       6     call p (e,m);
var g: T                              assume g = g?;
var G[M]                          8     g := G[m];
                                    else // m ≤ cur_lvl
// translation of                10     let saved_G = G[m+1..cur_lvl] in
// proc main () s                       let saved_g = g in
proc main ()                      12     let g? = ⋆ in
   done := false;                       g := G[m];
   s;                             14     G[m] := g?;
   done := true                         call p (e,m);
                                  16     assume g = g?;
// translation of g                     G[m+1..cur_lvl] := saved_G;
g                                 18     g := saved_g;
```

Figure 7. The sequential translation $(\!|P|\!)_{\mathrm{p}}$ of a yield-free single-buffer asynchronous program P with task priorities.

In the case of a single priority level, our asynchronous-to-sequential code translation $(\!|\cdot|\!)_{\mathrm{p}}$ of Figure 7 is identical to the no-delay depth-first scheduling sequentialization [7]. Each **post** statement is roughly translated to a **call** statement. Since each posted task t must execute after the completion of the currently executing task, at the point where t is called, the current valuation of the global variable `g` does not generally correspond to t's initial global valuation. The sequentialization thus introduces an auxiliary variable `G`, which holds at any point the global valuation encountered by the next-to-be-posted task. Initially, `G`'s value is guessed, and later verified to be the value reached by the initial task. Each time a new task t is posted, `G` is updated with a guess of the value reached by t and t begins with the previous value stored in `G` (Lines 12–14), and when t completes, we verify that the guessed value indeed matches the value reached by t (Line 16). The simulated execution corresponds to a depth-first traversal of the tree of tasks connected by the task-posting relation, in which each task executes atomically, to completion.

The presence of multiple priority levels makes our translation significantly more intricate. First, instead of a single auxiliary variable `G` storing the next-to-be-posted task's valuation, we must track that valuation *per priority level*, due to the additional task ordering constraints. Second, when a call to a higher-level posted task returns, we must update the global valuation of the currently-executing task to that reached by the higher-priority tasks (Line 8), rather than restoring previous global valuation (as in Line 18); this captures interruption. Third, calls to lower-level m_1 tasks t_1 must not overwrite the values stored in `G` for levels between m_1 and the current level $m_3 > m_1$, e.g., by posting additional tasks t_2 to level m_2 between m_1 and m_3. Doing so would simulate executions in which t_2 executes before other same-level tasks t_2' not yet posted by level m_3; this would be a breach in causality, since t_2' must in reality execute before t_1, and thus before t_2. Our translation prevents such inconsistent behavior by saving the `G` values for levels between m_1 and m_3 (Line 10), and restoring them upon return from calls corresponding to lower-level posts (Line 17). Finally, since even the

value encountered by the first-posted task is originally guessed, a simulated execution can only be considered valid when the main task completes and can validate the initial guess; we add the additional variable `done` to indicate whether the `main` task has completed. To simplify our translation, we assume priority-level m tasks post only tasks of priority at most $m+1$; posts to higher levels can be encoded by a chain of posts.

Formally, our *prioritized asynchronous to sequential translation*

$$\Theta_{\mathrm{P}} = \left\langle (\!|\cdot|\!)_{\mathrm{P}}, S_{\mathrm{P}}, f_{\mathrm{P}} \right\rangle$$

is the code transformation $(\!|\cdot|\!)_{\mathrm{P}}$ listed in Figure 7, along with the validity predicate S_{P} and reachability-fact map f_{P} defined as

- $S_{\mathrm{P}}(g_0, g_f) \stackrel{\text{def}}{=} \texttt{done}(g_f) = \texttt{true} \wedge \texttt{G[0]}(g_0) = \texttt{g}(g_f)$, and
- $f_{\mathrm{P}}(g_0, g_f) \stackrel{\text{def}}{=} \langle \texttt{g}(g_0), \texttt{G[0]}(g_f) \rangle$.

Given this correspondence, we reduce state-reachability of a single-buffer scheduler-dependent asynchronous program with priorities P to state-reachability of the sequential program $(\!|P|\!)_{\mathrm{P}}$.

Theorem 7 (Soundness)**.** *For all programs P, if $\langle g_0, g_f \rangle$ is a reachability fact of $(\!|P|\!)_P$ and $S_{\mathrm{P}}(g_0, g_f)$, then $f_{\mathrm{P}}(g_0, g_f)$ is a reachability fact of P.*

Note that we cannot state a completeness result for this final translation step, since the translation only encodes a deterministic schedule of task dispatching. However, when combined with the previous K-bounded yield-eliminating translation, and the assumption that the original program is reorderable, we do obtain completeness in the limit as K approaches infinity.

Theorem 8 (Completeness)**.** *For all reachability facts $\langle g_0, g_f \rangle$ of a reoderable single-buffer asynchronous program P there exists $K \in \mathbb{N}$ and a reachability fact $\langle g_0'', g_f'' \rangle$ of $(\!|(\!|P|\!)_Y^K|\!)_P$, and $\langle g_0', g_f' \rangle$ such that $\langle g_0', g_f' \rangle = f_{\mathrm{P}}(g_0'', g_f'')$, $\langle g_0, g_f \rangle = f_{\mathrm{Y}}^K(g_0', g_f')$, $S_{\mathrm{P}}(g_0'', g_f'')$, and $S_{\mathrm{Y}}^K(g_0', g_f')$.*

Finally, by choosing $K_1, K_2 \in \mathbb{N}$, composing our translations

$$\Theta_{\mathrm{M}}^{K_1} \circ \Theta_{\mathrm{Y}}^{K_2} \circ \Theta_{\mathrm{P}}$$

and gathering the results of Theorems 3–8, we have a sound algorithm for state-reachability of asynchronous programs with multiple prioritized task-buffers and arbitrary preemption, which is complete in the limit as both K_1 and K_2 approach infinity.

6. Implementation and Experience

We have implemented our sequentialization technique in a tool called AsyncChecker. It takes a concurrent program with assertions as input, written in the C programming language extended with the **post** primitive for spawning threads. This primitive allows us to easily model concurrency provided by, for example, the `pthread` library, or the Win32 API. The user is also given control on where to insert yields and zields; the default choice being that they are inserted before every access to a shared variable. AsyncChecker also takes two integers as input, which denote the zield budget (i.e., the bound used in Section 4) and the yield budget (i.e., the bound used in Section 5). It uses these budgets to sequentialize the program and then look for assertion violations within those budgets. When it finds an assertion violation, it displays the interleaved error trace. AsyncChecker has the unique capability of targetting one kind of bugs over another by changing the budgets: a high zield bound targets bugs that require inter-buffer interleavings, and a high yield bound targets bugs that require inter-task reorderings or preemptions. AsyncChecker uses Corral [15], an SMT-based model checker, as the sequential verifier. Appendix B details how we deal with assertions and error-trace generation in our sequentializations. We now give evidence that even though AsyncChecker uses an elaborate sequentialization, it is still a practical tool capable of finding real bugs using three different experiments.

Name	Outcome	Poirot (sec)	AsyncChecker (sec)
driver1	Correct	8.66	10.63
driver2	Buggy	4.42	3.44
driver3	Buggy	7.48	9.14
driver4	Correct	13.26	9.61
driver5	Buggy	6.1	5.2
driver6	Correct	37.57	77.1
driver7	Correct	44.43	86.56
driver8	Buggy	24.33	29.1
driver9	Buggy	31.5	33.9
driver10	Correct	23.64	25.4

Figure 8. Results on Poirot regressions.

First Experiment. Our first experiment is to compare AsyncChecker against other more mature tools on finding bugs in multi-threaded programs (with a single task-buffer and no priorities). This is to show: (1) Although AsyncChecker is a prototype implementation, it already competes well with existing tools on real programs; (2) one does not pay the cost of using multiple task-buffers or priorities unless the program uses these features. We compare AsyncChecker against Poirot [23], which is also based on a sequentialization technique [16, 17] that requires a thread round-robin bound, similar in spirit to AsyncChecker's yield budget. Poirot also uses Corral as the underlying sequential checker.

For benchmarks, we used Poirot's regression suite that consists of real device drivers. Some drivers have seeded bugs. The sizes of the drivers ranged from 500 to 700 lines of source code, with three threads that exercise different routines exposed by the driver. The results are shown in Fig. 8. We ran AsyncChecker with a zield budget of 1 and a yield budget equal to the round-bound used by Poirot. In each case, the outcome of Poirot and AsyncChecker (buggy or correct) was the same. AsyncChecker performed reasonably well in comparison to Poirot.

For programs with multiple task-buffers or prioritized tasks, there is no tool (other than AsyncChecker) for analyzing them, to the best of our knowledge. However, there is still an alternative to using AsyncChecker: one can encode the task-buffers and priority levels using shared memory and synchronization and then use Poirot on the resulting multithreaded program. We also implemented this approach (lets call it SyncE). It introduces shared state that counts the number of tasks in each buffer and at each priority level. Next, it inserts assume statements to prevent a task from running if the buffer size at a higher priority level is non-empty.[8]

Second Experiment. Consider the single-buffer program shown in Fig. 9. (We assume that there are implicit yields and zields between every two instructions.) The program has a single assertion in `bar` that checks the value of `x` against a parameter `N`. For any positive value of `N`, there is an execution that violates the assertion. Moreover, that execution necessarily has to alternate between priorities `0` and `1` for a total of `N` times. We ran both AsyncChecker and SyncE on this program with different values of `N`. In each case, we report the time taken by the tools to find the assertion violation under the best possible budget values.

The table in Fig. 9 shows the poor scalability of SyncE. The reason is that AsyncChecker can find the bug using a yield budget of 1, but Poirot (running underneath SyncE) requires a context-switch bound proportional to `N` because of SyncE's encoding of priorities. The increase in this bound causes exponential slowdown.

[8] Appendix C outlines this approach in detail using a code-to-code transformation.

```
var x: ℕ
var cont: 𝔹

proc bar()
  var t := x
  x := t + 1
  assert (x != N)
  if(⋆) then
     cont := true
```

```
proc main()
  x := 0; cont := true;
  call foo();

proc foo()
  if (cont and ⋆) then
     cont := false
     post 1 bar()
     post 0 foo()
```

N	1	2	3	4
SYNCE	1.5	2.1	13.6	384.2
ASYNCCHECKER	1.2	1.2	1.25	1.34

Figure 9. A single-buffer example and the running times (in seconds) of SYNCE and ASYNCCHECKER.

```
var x := 0

proc main2()
  var j := 0
  while(⋆) do
     assume(x = 2⋆j + 1)
     post 1 bar()
     j := j + 1
     assert(x != 2⋆N)
```

```
proc main1() {
  var i := 0;
  while(⋆) do
     assume(x = 2⋆i)
     post 1 bar()
     i := i + 1

proc bar()
  var t := x; x := t + 1;
```

N	1	2	3	4
SYNCE	1.6	5.1	168.9	> 500
ASYNCCHECKER	1.1	1.7	7.23	256.6

Figure 10. A multi-buffer example and the running times (in seconds) of SYNCE and ASYNCCHECKER.

The next example, shown in Fig. 10 has two task buffers with initial tasks `main1` and `main2`, respectively. Again, there are implicit yields and zields between every two instructions, except in `bar` that is supposed to execute yield free. This program requires switching between buffers each time the shared variable is incremented. Note that intra-buffer yielding is not necessary for this example—an observation that ASYNCCHECKER can exploit by setting the yield budget to 1 and only increasing the zield budget to find the bug. The results show that even though ASYNCCHECKER has an exponential dependence on the zield budget, it scales much better than SYNCE.

This experiment shows that an encoding of priorities and multiple buffers using shared memory may not fit well with the analyses that follows. This motivates our first-class treatment of such features in ASYNCCHECKER.

Third Experiment. Inspired by typical hardware-software interaction in Windows [19], we hand-coded a small C program (117 lines of code) that has two task-buffers (one for the hardware and one for the software) and three priority levels (`DEVICE_LEVEL`, `DISPATCH_LEVEL` and `PASSIVE_LEVEL`). Tasks running at level above `PASSIVE_LEVEL` do not yield. The driver, running on the software buffer signals the hardware to read data from a device. When the hardware finishes reading data, it raises an interrupt line (a shared variable). When tasks in the software buffer see that the interrupt line has been raised, they post the interrupt handler (ISR) at `DEVICE_LEVEL` (and immediately interrupt themselves). Two read requests from the driver can cause two hardware interrupts to fire, but, because the ISR runs at an elevated level, it cannot interleave with other instances of the ISR task. We seeded a bug in this program where some read request gets dropped without being processed by the driver.

This example is to show why one needs our model. (1) Inter-buffer interleaving is necessary to model the hardware-software interaction (namely, the interrupt mechanism), and (2) priorities are necessary as well: POIROT, which does not understand priorities, gets distracted and reports a data race in the ISR, which is incorrect. ASYNCCHECKER, on the other hand, takes 440 seconds to find the seeded bug (and is incapable of reporting the erroneous data race reported by POIROT). SYNCE takes too long to finish.

7. Related Work

Our model of asynchronous programs with prioritized task-buffers is inspired by and extends the classical single-buffer asynchronous programming model [9, 11, 26], and that with task-priorities considered once before [3]. Though Atig et al. [3] showed decidability of state-reachability in the asynchronous programming model with priorities, their decision procedure relies on reachability in Petri nets, for which the only known algorithms are extremely complex—they are non-primitive recursive. We build on this body of work by (1) adding multiple task-buffers, (2) demonstrating real-world examples which rely on task priorities and/or multiple task-buffers, and (3) giving a relatively practical parameterized under-approximating algorithm for state reachability, which incrementally explore more and more program behaviors as the parameter value increases, by reduction to state-reachability in sequential programs.

Our work closely follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called "sequentialization" [25] explored multi-threaded programs up to one context-switch between threads, and was later expanded to handle a parameterized amount of context-switches between a statically-determined set of threads executing in round-robin order [17]. La Torre et al. [13] later provided an alternate encoding better suited toward model-checking the resulting sequential program, and eventually extended the approach to handle programs parameterized by an unbounded number of statically-determined threads [14]. Shortly after, Emmi et al. [7] further extended these results to handle an unbounded amount of dynamically-created tasks, which besides applying to multi-threaded programs, naturally handles asynchronous event-driven programs [9, 11, 26]. Bouajjani et al. [4] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget. Though the latter two of these sequentializations are applicable to asynchronous programs dynamically creating an unbounded number of tasks, they do not account for task priorities, nor multiple task buffers. Though Kidd et al. [12] have demonstrated a priority-aware sequentialization, their reduction assumes a fixed number of statically-determined tasks, and does not account for multiple task buffers.

8. Conclusion

We have introduced a formal model of asynchronous programs with multiple prioritized task-buffers, which closely captures the concurrency present in many real-world asynchronous systems. Though program analysis in our model is complex, we propose an incremental approximate analysis algorithm by reduction to sequential program analysis. The parameters $K_1, K_2 \in \mathbb{N}$ to our sequential reduction restrict, resp., the amount of inter-buffer interleaving and intra-buffer task orderings explored; in the limit as K_1 and K_2 approach infinity, our reduction explores all program behaviors. We demonstrate that this reduction is relatively easy to implement, and by using off-the-shelf sequential analysis tools is able to discover concurrency errors, without reporting spurious

errors due to modeling imprecision, in asynchronous device driver code in the Windows kernel.

References

[1] Apple Computer Inc. Apache GCD multi-processing module. `http://libdispatch.macosforge.org/trac/wiki/apache`.

[2] Apple Computer Inc. Grand central dispatch (GCD) reference. `http://developer.apple.com/library/mac/#documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html`.

[3] M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *FSTTCS '08: Proc. IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *LIPIcs*, pages 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[4] A. Bouajjani, M. Emmi, and G. Parlato. On sequentializing concurrent programs. In *SAS '11: Proc. 18th International Symposium on Static Analysis*, volume 6887 of *LNCS*, pages 129–145. Springer, 2011.

[5] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[6] R. Dahl. Node.js: Evented I/O for V8 JavaScript. `http://nodejs.org/`.

[7] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422. ACM, 2011.

[8] J. Esparza. Decidability and complexity of Petri net problems - an introduction. In *Petri Nets: Proc. Dagstuhl Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *LNCS*, pages 374–428. Springer, 1998.

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

[10] I. Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. `http://dev.w3.org/html5/spec/Overview.html`.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S. Lahiri, A. Lal, and S. Qadeer. Corral: A whole program analyzer for boogie. In *BOOGIE Workshop*, 2011.

[16] 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.

[17] 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.

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

[19] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey. An automata-theoretic approach to hardware/software co-verification. In *FASE*, pages 248–262, 2010.

[20] Microsoft Inc. Parallel programming in the .NET framework. `http://msdn.microsoft.com/en-us/library/dd460693.aspx`.

[21] Microsoft Inc. Windows kernel-mode driver architecture. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff557560(v=VS.85).aspx`.

[22] 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.

[23] Poirot: The Concurrency Sleuth. `http://research.microsoft.com/en-us/projects/poirot/`.

[24] 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.

[25] 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.

[26] 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.

[27] M. Wilcox. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *LCA '03: Proc. 4th Linux Conference Australia*, 2003.

A. Syntactic Sugar

The following syntactic extensions are reducible to the original program syntax of Section 3. 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 $\texttt{x}_1\texttt{:}\ T_1\texttt{, ..., x}_j\texttt{:}\ T_j$ can be encoded with a single record-typed variable $\texttt{x:}\ T$, where T is the record type

$$\texttt{\{ f}_1\texttt{:}\ T_1\texttt{, ..., f}_j\texttt{:}\ T_j\ \texttt{\}}$$

and all occurrences of $\texttt{x}_i$ are replaced by $\texttt{x.f}_i$. 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 $\textbf{var l':}\ T$ to a procedure p can be encoded by adding `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 `l'` begins correctly (un)initialized.

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

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

Tuples. Assignments `(x1, ..., xj) :=` e to a tuple of variables `x1` ... `xj` are encoded by the sequence

```
let r: { f1: T1, ..., fj: Tj } = e in
x1 := r.f1; ...; xj := r.fj
```

where `r` is a fresh variable. A tuple expression `(x1, ..., xj)` occurring in a statement s is encoded as

```
let r: { f1: T1, ..., fj: Tj } = { f1 = x1, ...,
    fj = xj } in
s[r/(x1, ..., xj)]
```

where `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 `xi` 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 `xi` with the `_` variable—see the "unused values" desugaring.

Arrays. Finite arrays with j elements of type T can be encoded as records of type `{ f1: T, ..., fj: T }`, where `f1` ... `fj` are fresh names. Occurrences of terms `a[i]` are replaced by `a.fi`, and array-expressions `[ e1, ..., ej ]` are replaced by record-expressions `{ f1 = e1, ..., fj = ej }`.

B. Asserts and Error Traces

Assertions. The program transformations presented in this paper preserve reachability facts, which talk about starting and end state of a program execution. In order to check for assertions in the program, we first map them to reachability facts as follows: when an asserted condition is violated, the program is intrumented to raise a flag and throw an exception. The exception has the effect of aborting the current as well as all pending tasks. If we see that the exception flag been raised in the end state of an execution, then we know that an assertion must have been violated in that execution. Thus, our implementation asserts that the exception flag is not set in the sequential program produced by our technique.

Error Traces. When the sequential verifier finds a bug in the sequential program produced by our technique, we still need to map the bug back to an execution of the original concurrent program. We follow a general approach to solve this problem, which is easily adaptable to other sequentializations as well. First, we introduce a primitive in our model called **print** that takes a single variable as argument. We assume that when the sequential verifier finds an error trace (in the sequential program), and it passes through some **print** statements, then it prints the values of their arguments in the order they appear on the trace.

Next, we introduce an extra integer variable called `traceCnt` in the original program that is initialized to `0`. Finally, at the beginning of each task and after every **yield**, **zield** and **post** statement, we insert the code "`loc := traceCnt;` **print**`(loc);` `traceCnt++`", where `loc` is a fresh local variable. Because the print statement only operates on local state, it will be left untouched by our sequentialization. When the sequential verifier reports an error trace, one simply has to follow the sequence of printed values to reconstruct the interleaving in the error trace.

C. Encoding buffers and priorities using shared memory

This section describes one approach for encoding a multi-buffer and multi-priority program as a single-buffer and single-priority program using shared-memory and synchronization. The encoding is a code-to-code translation, shown in Figure 11. The translation assumes that at any point in the program's execution, the following is known about the currently executing task: `curr_level` is its priority level; `curr_buff` is the buffer it came from, `curr_tid` is its task identifier. (We assume that each task is associated with a unique task identifier, which is distinct from 0.)

The translation introduces three extra shared variables with the following intention: `BufferSize[b][l]` records the number of pending tasks in buffer number `b` at priority level `l`; `Interrupted[b][l]` records the identifier of the task of level `l` and buffer `b` that last got interrupted by a higher priority task (if any); `Switched[b]` records the identifier of the task of buffer `b` that last got preempted by another buffer (if any). The **assume** statements introduced by the translation are the ones that rule out infeasible executions. The other assignments do the appropriate book-keeping for the extra variables introduced.

Additional declarations
var `BufferSize:` $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
var `Interrupted:` $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
var `Switched:` $\mathbb{N} \to \mathbb{N}$

Additional initialization
`forall b,l. BufferSize[b][l] := 0`
`forall b,l. Interrupted[b][l] := 0`
`forall b. Switched[b] := 0`

Statement/Expr	Translation
yield	**yield**; **assume** (forall l > curr_level. BufferSize[curr_buff][l] == 0); **assume** (Interrupted[curr_buff][curr_level] == 0); **assume** (Switched[curr_buff] == 0);
zield	Switched[curr_buff] := curr_tid; **yield**; Swicthed[curr_buff] := 0;
return from `a task`	BufferSize[curr_buff][curr_level] --; **return**;

post m p e	**if** m > curr_level **then**
	  BufferSize[curr_buff][m]++;
	  Interrupted[curr_buff][curr_level] := curr_tid;
	  post p e
	  yield;
	  assume (forall l > curr_level. BufferSize[curr_buff][l] == 0);
	  Interrupted[curr_buff][curr_level] := 0;
	else
	  BufferSize[curr_buff][m] ++;
	  post p e

Figure 11. Semantics-preserving encoding of multi-buffer and multi-priority programs to ones with a single buffer and a single priority.