On Sequentializing Concurrent Programs

Ahmed Bouajjani, Michael Emmi, Gennaro Parlato

On Sequentializing Concurrent Programs

Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato

LIAFA, Université Paris Diderot, France
{abou,mje,gennaro}@liafa.jussieu.fr

Abstract. We propose a general framework for compositional under-approximate concurrent program analyses by reduction to sequential program analyses—so-called *sequentializations*. We notice the existing sequentializations—based on bounding the number of execution contexts, execution rounds, or delays from a deterministic task-schedule—rely on three key features for scalable concurrent program analyses: (i) reduction to the *sequential* program model, (ii) *compositional* reasoning to avoid expensive task-product constructions, and (iii) *parameterized* exploration bounds. To understand how those sequentializations can be unified and generalized, we define a general framework which preserves their key features, and in which those sequentializations are particular instances. We also identify a most general instance which considers more executions, by composing the rounds of different tasks in any order, restricted only by the unavoidable program and task-creation causality orders. In fact, we show this general instance is fundamentally more powerful by identifying an infinite family of state-reachability problems (to states $g_1, g_2, \ldots$) which can be answered precisely with a fixed exploration bound, whereas the existing sequentializations require an increasing bound k to reach each g_k. Our framework applies to a general class of shared-memory concurrent programs, with dynamic task-creation and arbitrary preemption.

1 Introduction

Concurrent program analysis is difficult due to the high computational complexity that arises when considering every intricate task interleaving. To avoid such high computational complexity, bounded (underapproximating) exploration is emerging as a general technique. In general, one characterizes a subset of the concurrent program semantics by a bounding parameter k. As k is increased we explore more behaviors, at the expense of more computational resources; in the limit we explore every behavior. A bounded exploration technique is effective when useful information (e.g., the presence of bugs) is obtained by spending a relatively small amount of computational resources (i.e., using low values of k).

By characterizing a subset of the concurrent program semantics by a bounding parameter (e.g., bounded context-switch [21]), it is often possible to reduce concurrent program exploration to sequential program exploration [22, 19, 14, 16, 7, 13]—essentially by constructing an equivalent program without concurrency primitives (e.g., task creation, preemption). Such *sequentializations* are desirable, both practically and theoretically, for several reasons. First, they reduce the

exploration problem of any concurrent program to the well-understood model of sequential programs, and are not limited by finite-data programs, per se. Here several practical verification algorithms exist by way of finite-data software model checking [23, 4, 3, 15]), (infinite-state) fixed point computations [5, 24, 11], and (e.g., SMT-based) bounded software verification algorithms [6, 17]. Second, they compute the behavior of each concurrent task compositionally—i.e., without considering the local-state valuations of other tasks. Third, they enable incremental analysis, trading-off computing resources for the amount of explored behaviors, by varying a bounding parameter $k \in \mathbb{N}$. (The parameter k determines the (asymptotic) budget of the sequential program exploration. In practice, the reductions add $\mathcal{O}(k)$ variables to the resulting sequential program, resulting in an increasingly-expensive (in k) sequential program exploration.) Indeed these sequentialization-based analyses have been applied to discover (in some cases previously-unknown) bugs in device drivers [19, 14, 16, 18, 7].

Our principal aim in this work is to develop a theory of parameterized, compositional, concurrent-to-sequential program analysis reductions. Specifically, we make the following contributions:

- To identify the fundamental mechanisms *enabling* compositional sequentializations, thus formulating a framework in which to define them (Section 3).
- To formulate a most general sequentialization in our framework which expresses as many concurrent behaviors as possible (Section 4), while maintaining compositionality, and without extending the class of programs in which tasks are originally written—e.g., by adding unbounded counters.
- To classify the existing sequentializations in our framework, and compare them w.r.t. the behaviors explored for their bounding parameters (Section 5).

Besides enlightening the mechanisms which enable sequential reduction, we believe our gained insight can guide further advances, for instance by considering other restrictions to our framework with efficient encodings.

Using the three features discussed above ((i) reduction to sequential programs, (ii) compositionality, and (iii) parameterization), the existing sequentializations work by characterizing each concurrent task by an interface of k global-state valuation pairs (where k is the bounding parameter). These k global-state valuation pairs represent a computation of a task which is interrupted $k-1$ times—the valuations give the initial and final global-states per execution "round." Each task's interface is then computed—in isolation from other tasks—by beginning each round from the guessed global-state valuation, performing a sequence of sequential program steps, then eventually moving to the next round, by updating the current global-state to the next-guessed valuation. Contiguous interfaces (i.e., where the final global-state valuations of one matches the initial valuations of the other) are then glued together to form larger computations. Intuitively, this interface composition builds executions according to a round-robin schedule of k rounds; a complete execution is formed when each i^{th} final global-state of the last task's interface matches the $(i+1)^{\text{st}}$ initial global-state of the first's. When there are a fixed number of statically-created tasks, interfaces are simply composed in task-identifier order. In the more general case, with *dynamically*

created tasks, interfaces are composed in a depth-first preorder over the ordered task-creation tree. (Note by viewing the task-identifier order as the task-creation order, the dynamic case subsumes the static.)

Thus, besides features/constraints (i), (ii), and (iii)—which we feel are desirable, and important to maintain for efficiency and scalability of the resulting program analyses—the existing sequentializations are constrained by: (iv) round-robin executions (in depth-first order on the ordered task-creation tree). Though other schedules can be simulated by increasing the (round) bounding parameter k when the number of tasks is bounded, for a fixed value of k, only k-round round-robin schedules are explored.

The most general instance of the framework we propose in Section 4 subsumes and extends the existing round-robin based (i.e., context- and delay-bounded) sequentializations. As in these previous approaches, we restrict ourselves to explorations that are (i) sequential, (ii) compositional, and (iii) parameterized, in order to compute task interfaces over k global-state valuation pairs. However, for the same analysis budget[1] (i.e., the bounding parameter k), the general instance expresses many more concurrent behaviors, essentially by relaxing the round-robin execution order, and decoupling each task from any global notion of "rounds." We consider that each task's interface is constructed by composing the rounds of tasks it has created in *any* order that respects program order, and task-creation causality order—i.e., a task is not allowed to execute before a subsequent preemption of the task that created it. Thus the simulated concurrent execution need not follow a round-robin schedule, despite the prescribed task traversal order; the possible inter-task interleavings are restricted only by the constraint that at most k sub-task round summaries can be kept at any moment—indeed this constraint is necessary to encode the resulting sequentialization with a bounded number of additional (finite-domain) program variables.

In fact, the most general instance of our framework is fundamentally more expressive than the existing sequentializations, since there are infinite sequences of states (e.g., $g_1, g_2, \ldots$) which can be reached with a fixed exploration bound, whereas the existing sequentializations require an increasing bound k to reach each g_k. This gain in expressiveness may or may not be accompanied by an analysis overhead. For enumerative program analysis techniques (e.g., model checking, systematic testing), one may argue that considering more interleavings will have a negative impact on scalability. This argument is not so clear, however, for symbolic techniques (e.g., SMT-based verification, infinite-state fixed-point computations), which may be able to reduce reasoning over very many interleavings to very few, in practice [19]. Although many device driver bugs have been discovered within few interleavings [20], reducing the number of interleavings could indeed cause missed bugs in other settings. Furthermore, our hope is that enlightening the mechanisms behind sequentialization will lead to the discovery of other succinctly-encodable instances of compositional sequentialization.

[1] See Section 4 for complexity considerations.

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

Fig. 1. The grammar of asynchronous programs. Each program P declares a single type-T global variable $\texttt{g}$, and a sequence of procedures named $p_1 \ldots p_n \in \mathsf{Procs}^*$. Each procedure p has single type-T parameter $\texttt{l}$, and a top-level statement (i.e., the procedure body) denoted s_p. This program syntax is made simple only to simplify presentation; various extensions—e.g., to multiple global and local variables—can be encoded by varying the type T; see Appendix A.

2 Concurrent Programs

We consider a simple but general concurrent programming model in the style of single-threaded event-driven programs. (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*.[2]) In this model, control begins with an initial task, which is essentially a sequential program that can read from and write to global (i.e., shared) storage, and *post* additional tasks (each **post** statement specifies a procedure name and argument) to an initially empty *task buffer* of pending tasks. When control returns to the dispatcher, either when a task *yields* control—in which case it is put back into the task buffer—or completes its execution, the dispatcher picks some task from the task buffer, and transfers control to it; when there are no pending tasks to pick, the program terminates. This programming model is powerful enough to model concurrent programs with arbitrary preemption and synchronization, e.g., by inserting a **yield** statement before every shared variable access.

Let Procs be a set of procedure names, Vals a set of values containing $\texttt{true}$ and $\texttt{false}$, and T the sole type of values. The grammar of Fig. 1 describes our language of *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 Exprs contains Vals and the *(nullary) choice operator* $\star$. The set of program statements s is denoted Stmts. A *sequential program* is an asynchronous program which contains neither **post** nor **yield** statements.

A *(procedure-stack) frame* $\langle \ell, s \rangle$ is a valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable $\texttt{l}$, along with a statement s to be executed. (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.) A *task* w is a sequence of frames representing a procedure stack, and the set $(\mathsf{Vals} \times \mathsf{Stmts})^*$ of tasks is denoted Tasks. An *(asynchronous)*

[2] As our development is independent from architectural particularities, "tasks" may correspond to threads, processes, asynchronous methods, etc.

$$\textsc{Post}\ \frac{\ell' \in e(g,\ell) \qquad w' = \langle \ell', s_p \rangle}{\langle g, \langle \ell, S[\mathbf{post}\ p\ e] \rangle\, w, m\rangle \to^{\mathrm{a}}_P \langle g, \langle \ell, S[\mathbf{skip}] \rangle\, w, m \cup \{w'\}\rangle} \qquad \textsc{Dispatch}\ \frac{w \in m}{\langle g, \varepsilon, m\rangle \to^{\mathrm{a}}_P \langle g, w, m \setminus \{w\}\rangle}$$

$$\textsc{Complete}\ \frac{}{\langle g, \langle \ell, S[\mathbf{return}\ e] \rangle, m\rangle \to^{\mathrm{a}}_P \langle g, \varepsilon, m\rangle} \qquad \textsc{Yield}\ \frac{w' = \langle \ell, S[\mathbf{skip}] \rangle\, w}{\langle g, \langle \ell, S[\mathbf{yield}] \rangle\, w, m\rangle \to^{\mathrm{a}}_P \langle g, \varepsilon, m \cup \{w'\}\rangle}$$

Fig. 2. The transition relation $\to^{\mathrm{a}}_P$ for an asynchronous program P is given by combining the transitions above with the sequential transition relation $\to^{\mathrm{s}}_P$.

configuration $c = \langle g, w, m\rangle$ is a global-variable valuation $g \in \mathsf{Vals}$ along with a task $w \in \mathsf{Tasks}$, and a task buffer multiset $m \in \mathbb{M}[\mathsf{Tasks}]$.

To define the transition relation between configurations, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ for expressions without program variables, such that $[\![\star]\!]_{\mathrm{e}} = \mathsf{Vals}$. For convenience, we define $e(g,\ell) \stackrel{\text{def}}{=} [\![e[g/\mathtt{g}, \ell/\mathtt{l}]]\!]_{\mathrm{e}}$ —since $\mathtt{g}$ and $\mathtt{l}$ are the only variables, the expression $e[g/\mathtt{g}, \ell/\mathtt{l}]$ has no free variables. To select the next-scheduled statement in a configuration, we define a *statement context* S as a term derived from the grammar $S ::= \diamond \mid S;\ s$, and write $S[s]$ for the statement obtained by substituting a statement s for the unique occurrence of $\diamond$ in S.

The transition relation $\to^{\mathrm{a}}_P$ of an asynchronous program P is defined in Fig. 2 as a set of operational steps on configurations. The transition relation $\to^{\mathrm{s}}_P$ for the sequential program statements (see Appendix C) is standard. The Post rule gives a newly-created task to the task buffer, while the Yield rule gives the currently-executing task to the task buffer. The Dispatch rule chooses some task from the buffer to execute, and the Complete rule disposes a completed task.

A configuration $\langle g, \langle \ell, s\rangle, \emptyset\rangle$ is called *initial*, and is *sequential* when s does not contain **post** (nor **yield**) statements. An *(asynchronous) execution of a program P (from c_0 to c_j)* is a configuration sequence $h = c_0 c_1 \dots c_j$ where

- c_0 is initial, and
- $c_i \to^{\mathrm{a}}_P c_{i+1}$ for $0 \le i < j$.

We say a configuration $c = \langle g, w, m\rangle$ (alternatively, the global value g) is *reachable in P (from c_0)* when there exists an execution of P from c_0 to c. The *asynchronous semantics* of P, written $\{\!|P|\!\}_{\mathrm{a}}$, maps initial configurations to reachable global values, i.e., $\{\!|P|\!\}_{\mathrm{a}}(c_0) = g$ if and only if g is reachable in P from c_0. When P is a sequential program, the *sequential semantics* of P, written $\{\!|P|\!\}_{\mathrm{s}}$, maps initial sequential configurations to reachable global values.

3 Compositional Semantics

Here we define a *compositional* semantics for asynchronous programs on which to base our reduction to sequential programs. To do so, we characterize each posted task by an *interface* exposing only global-state valuations to other tasks. Each interface summarizes not only the computation of a single task, but also the

computations of all descendants of tasks it has posted. In particular, an interface is a sequence $\langle g_1, g_1' \rangle \ldots \langle g_k, g_k' \rangle$ of global-state valuation pairs summarizing a computation which is interrupted $k-1$ times; the computation begins with global state g_1, is interrupted by an external task at global state g_1', is resumed at g_2, etc. We call the computation summarized by each pair $\langle g_i, g_i' \rangle$ *the* i^{th} *round*. A larger computation is then formed from a collection of task interfaces, by gluing together contiguous rounds (e.g., summarized by $\langle g_1, g_2 \rangle$ and $\langle g_2, g_3 \rangle$), while maintaining the order between rounds from the same interface.

We construct interfaces inside a data-structure called a *summary bag*. Besides the sequence $\mathcal{B}_{\text{ex}}$ of rounds which will be exported as the current task's interface (see Fig. 3a), the bag maintains a collection $\mathcal{B}_{\text{im}}$ of interfaces imported from posted tasks (see Fig. 3c). At any yield-point, the current task can begin a new round, by *guessing*[3] the global-state valuation that will begin the next round (see Fig. 3b), or, if the current global-state valuation matches the start of the first unconsumed round from some imported interface (as does $\langle a, b \rangle$ in Fig. 3d), the current task can consume that round and update the current global state; this amounts to interleaving the round of a posted task at the current control point.

With this view of recursive interface construction—interfaces are constructed from interfaces of sub-tasks—the reduction to sequential programs is nearly straightforward. To compute the imported summary of a posted task, we translate the **post** statement into a procedure call which returns the computed interface for the posted task. At yield points, we will repeatedly, nondeterministically choose to begin a new round, consume a round from an imported interface, or step past the **yield** statement. For the moment the only obstacle is how to store the unbounded summary bag; we address this issue in Section 4.

To define the compositional semantics we formalize the summary-bag operations. A *summary bag* $\mathcal{B} = \langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}} \rangle$ pairs the round sequence $\mathcal{B}_{\text{ex}}$ to be exported from the current task, along with a collection $\mathcal{B}_{\text{im}}$ of round sequences imported from sub-tasks. The *empty bag* $\mathcal{B}_\emptyset = \langle \varepsilon, \emptyset \rangle$ is an empty sequence paired with an empty collection. The operations to add the current round ($\oplus$), import a sub-task interface ($\odot$), and consume a sub-task round ($\ominus$) are defined as

$$\begin{aligned}
\langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}} \rangle \oplus \langle g, g' \rangle &\stackrel{\text{def}}{=} \langle \mathcal{B}_{\text{ex}} \cdot \langle g, g' \rangle, \mathcal{B}_{\text{im}} \rangle, && \text{add round} \\
\langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}} \rangle \odot \mathcal{B}_{\text{ex}}{}' &\stackrel{\text{def}}{=} \langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}} \cup \mathcal{B}_{\text{ex}}{}' \rangle, && \text{import interface} \\
\langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}} \rangle \ominus \langle g, g' \rangle &\stackrel{\text{def}}{=} \langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}}{}' \rangle && \text{consume round,}
\end{aligned}$$

where $\mathcal{B}_{\text{im}}{}'$ is obtained by removing $\langle g, g' \rangle$ from the head of a sequence in $\mathcal{B}_{\text{im}}$, and $\langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}} \rangle \ominus \langle g, g' \rangle$ is undefined if $\langle g, g' \rangle$ is not at the head of $\mathcal{B}_{\text{im}}$.

We define the compositional transition relation of asynchronous programs by augmenting the sequential transitions $\rightarrow_P^{\text{s}}$ with a new set of transitions for the asynchronous control statements, over a slightly different notion of configuration. In order to build interfaces, each configuration carries the global-state valuation at the beginning of the current round (denoted g_0 below), as well as the current

[3] An enumerative analysis algorithm can handle "guessing" by attempting every reached global valuation; a symbolic algorithm just creates a new symbol.

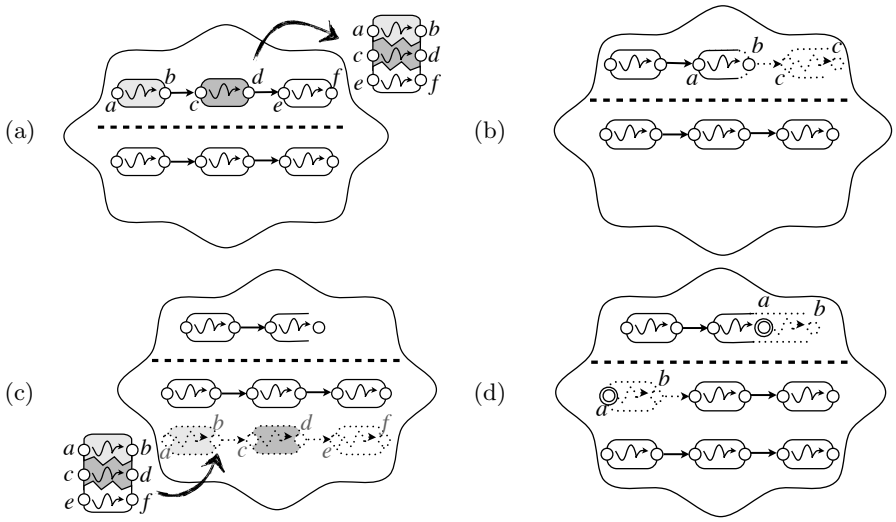


Fig. 3. The summary bag operations. Circles represent global valuations (double circles equivalences), and are paired together in rounded rectangle round summaries. Interfaces are drawn as stacked round summaries, and shading is used only to easily identify summaries. (a) Export a constructed interface, (b) begin a round by finalizing the current round $\langle a, b\rangle$, and guessing a valuation c to begin the next, (c) import sub-task interface, (d) interleave the first unconsumed round $\langle a, b\rangle$ of a sub-task, updating the current global state from a to b. The round-summaries $\mathcal{B}_{\text{ex}}$ to be exported appear above the dotted line, and the collection $\mathcal{B}_{\text{im}}$ of imported sub-task interfaces appears below.

valuation. A *(compositional) configuration* $c = \langle g_0, g, w, \mathcal{B}\rangle$ is an *initial* valuation $g_0 \in \mathsf{Vals}$ of the global variable g, along with a *current* valuation $g \in \mathsf{Vals}$, a task $w \in \mathsf{Tasks}$, and a summary bag $\mathcal{B}$.

Fig. 4 lists the transitions $\rightarrow^{\text{c}}_P$ for the asynchronous control statements. The NextRound rule begins a new round with a guessed global-state valuation, and the SubTask rule collects the posted task's interface. The task-summarization relation $p\ \ell' \rightsquigarrow \mathcal{B}_{\text{ex}}$ used in the SubTask rule holds when the task $\langle \ell', s_p\rangle$ can execute to a yield, or return, point having constructed the interface $\mathcal{B}_{\text{ex}}$—i.e., when there exists $g, g_0, g_1 \in \mathsf{Vals}$, $w \in \mathsf{Tasks}$, and $\mathcal{B}_{\text{im}}$ such that $\langle g, g, \langle \ell', s_p\rangle, \mathcal{B}_\emptyset\rangle \rightarrow^{\text{c}*}_P \langle g_0, g_1, w, \langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}}\rangle\rangle$, where w is of the form $\langle \ell, S[\textbf{yield}]\rangle\, w'$ or $\langle \ell, S[\textbf{return}\ e]\rangle$. The Interleave rule executes a round imported from some posted task, and finally, the Resume rule simply steps past the **yield** statement.

A configuration $\langle g, g, \langle \ell, s\rangle, \mathcal{B}_\emptyset\rangle$ is called *initial*. A *(compositional) execution of a program P (from c_0 to c_j)* is a configuration sequence $h = c_0 c_1 \ldots c_j$ where

- c_0 is initial, and
- $c_i \rightarrow^{\text{c}}_P c_{i+1}$ for $0 \le i < j$.

A compositional execution describes the progression of one task only; progressions of sub-tasks are considered recursively as separate executions to compute

$$\textsc{NextRound}\quad \frac{\mathcal{B}' = \mathcal{B} \oplus \langle g_0, g\rangle \qquad g' \in \mathsf{Vals}}{\langle g_0, g, \langle \ell, S[\mathbf{yield}]\rangle\, w, \mathcal{B}\rangle \to^{\mathrm{c}}_P \langle g', g', \langle \ell, S[\mathbf{yield}]\rangle\, w, \mathcal{B}'\rangle}$$

$$\textsc{SubTask}\quad \frac{\ell' \in e(g,\ell) \qquad p\ \ell' \rightsquigarrow \mathcal{B}_{\mathrm{ex}} \qquad \mathcal{B}' = \mathcal{B} \odot \mathcal{B}_{\mathrm{ex}}}{\langle g_0, g, \langle \ell, S[\mathbf{post}\ p\ e]\rangle\, w, \mathcal{B}\rangle \to^{\mathrm{c}}_P \langle g_0, g, \langle \ell, S[\mathbf{skip}]\rangle\, w, \mathcal{B}'\rangle}$$

$$\textsc{Interleave}\quad \frac{\mathcal{B}' = \mathcal{B} \ominus \langle g, g'\rangle}{\langle g_0, g, \langle \ell, S[\mathbf{yield}]\rangle\, w, \mathcal{B}\rangle \to^{\mathrm{c}}_P \langle g_0, g', \langle \ell, S[\mathbf{yield}]\rangle\, w, \mathcal{B}'\rangle}$$

$$\textsc{Resume}\quad \frac{}{\langle g_0, g, \langle \ell, S[\mathbf{yield}]\rangle\, w, \mathcal{B}\rangle \to^{\mathrm{c}}_P \langle g_0, g, \langle \ell, S[\mathbf{skip}]\rangle\, w, \mathcal{B}\rangle}$$

Fig. 4. The compositional transition relation $\to^{\mathrm{c}}_P$ for an asynchronous program P is given by combining the transitions above with the sequential transition relation $\to^{\mathrm{s}}_P$.

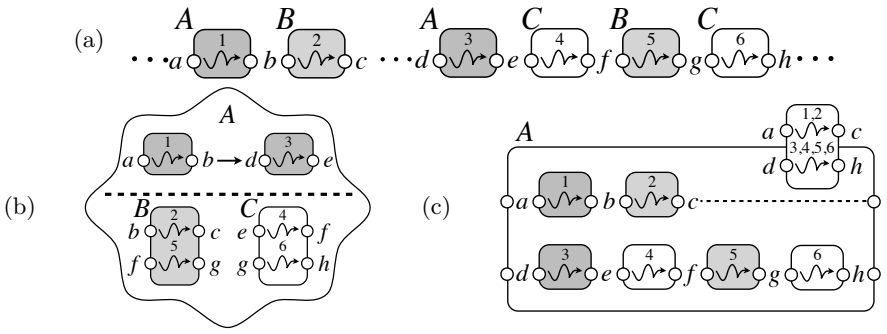


Fig. 5. Simulating an asynchronous execution (a) as a compositional execution, where task A posts B and C, then is interrupted by B, then is eventually re-dispatched, and upon completion is followed directly by C, which is interrupted by B before completing. (b) The bag used to reconstruct A's interface, and (c) the constructed interface for A.

the task-summarization relation $\rightsquigarrow$. We say a configuration $c = \langle g_0, g, w, \mathcal{B}\rangle$ (alternatively, the global value g) is *reachable in P (from c_0)* when there exists an execution from c_0 to c, without using the NextRound rule.[4] The *compositional semantics* of P, written $\{\!|P|\!\}_{\mathrm{c}}$, maps initial configurations to reachable global values, i.e., $\{\!|P|\!\}_{\mathrm{c}}(c_0) = g$ if and only if g is reachable in P from c_0.

Although their definitions are wildly different, the compositional and asynchronous semantics are equivalent. To see that every asynchronous execution h has a corresponding compositional execution, consider, inductively, how to build the interface summarizing the projection of h a given task's sub-task segments. Consider the task A of Fig. 5 which posts B and C. To simulate the asynchronous (sub-) execution of Fig. 5a, A builds an interface with two uninterruptible rounds (see Fig. 5c): the first sequencing segments 1 and 2, and the second sequencing 3, 4, 5, and 6. To build this interface, A must import two-round interfaces from B and C each; note that each round of B and C may recursively contain many interleaved segments of sub-tasks.

[4] Disallowing use of the NextRound rule in the top-level execution ensures that unchecked guessed global-state valuations are not considered reachable.

Theorem 1. *The compositional semantics and asynchronous semantics are identical, i.e., for all programs P we have $\{\!|P|\!\}_{\mathrm{c}} = \{\!|P|\!\}_{\mathrm{a}}$.*[5]

The proof to this theorem appears in Appendix B.

4 Bounded Semantics

As earlier sequentializations have done by constraining the number of rounds [19, 14], by constraining the size of the summary bag, we can encode the bag contents with a fixed number of additional variables in the resulting sequential program. Since we are interested in exploring as many behaviors as possible with a limited-size bag, an obvious point of attention is *bag compression*. Thus we define an additional operation to free a space in the bag by merging two contiguous (w.r.t. the global valuation) summaries, essentially removing the possibility for future reordering between the selected segments. In doing so, we must maintain causal order by ensuring the bag remains acyclic, and what was before a collection $\mathcal{B}_{\mathrm{im}}$ of summary sequences imported from sub-tasks now becomes a directed acyclic graph (DAG) of summaries. To maintain program order in the presence of merging, summaries are now consumed only from the roots of $\mathcal{B}_{\mathrm{im}}$.

The *size* $|\mathcal{B}| \stackrel{\text{def}}{=} |\mathcal{B}_{\mathrm{ex}}| + |\mathcal{B}_{\mathrm{im}}|$ of $\mathcal{B}$ is the sum of the length of the sequence $\mathcal{B}_{\mathrm{ex}}$ and the number of nodes of the DAG $\mathcal{B}_{\mathrm{im}}$. The bag simplification operation $\langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \rangle \rhd \langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}}{}' \rangle$ obtains $\mathcal{B}_{\mathrm{im}}{}'$ from $\mathcal{B}_{\mathrm{im}}$ by merging two nodes n and n', labelled, resp., by $\langle g_1, g_2 \rangle$ and $\langle g_2, g_3 \rangle$, such that either

(a) there is no directed path from n to n' in $\mathcal{B}_{\mathrm{im}}$, or
(b) there is an edge from n to n' in $\mathcal{B}_{\mathrm{im}}$

(see Fig. 6); in either case the merged node is labelled $\langle g_1, g_3 \rangle$. Note that when $\mathcal{B} \rhd \mathcal{B}'$ we have $|\mathcal{B}'| = |\mathcal{B}| - 1$. Though we could simulate this merge operation in the (unbounded) compositional semantics, by eventually consuming consecutively n and n' into the exported summary list, the merge operation allows to eagerly express the interleaving—though disallows any subsequent interleaving between n and n'. Merging summaries eagerly fixes a sequence of inter-task execution segments, trading the freedom for external interleaving (which may have uncovered additional, potentially buggy, behaviors) for an extra space in the bag.

We define the *k-bounded compositional semantics of P*, written $\{\!|P|\!\}^k_{\mathrm{c}}$, by restricting the compositional semantics $\{\!|P|\!\}_{\mathrm{c}}$ to executions containing only configurations $\langle g_0, g, w, \mathcal{B} \rangle$ such that $|\mathcal{B}| \leq k$, and adding the additional transition

$$\textsc{Compress}\ \frac{\mathcal{B} \rhd \mathcal{B}'}{\langle g_0, g, w, \mathcal{B} \rangle \rightarrow^{\mathrm{c}}_P \langle g_0, g, w, \mathcal{B}' \rangle}.$$

As we increase k, the set of k-bounded semantics grows monotonically, and in the limit, the k-bounded semantics *is* the compositional semantics. For two functions $f, g : A \to \wp(B)$, we write $f \subseteq g$ when for all $a \in A$, $f(a) \subseteq g(a)$.

[5] We consider initial configurations $\langle g, \langle \ell, s \rangle, \emptyset \rangle$ and $\langle g, g, \langle \ell, s \rangle, \mathcal{B}_\emptyset \rangle$ as equal.

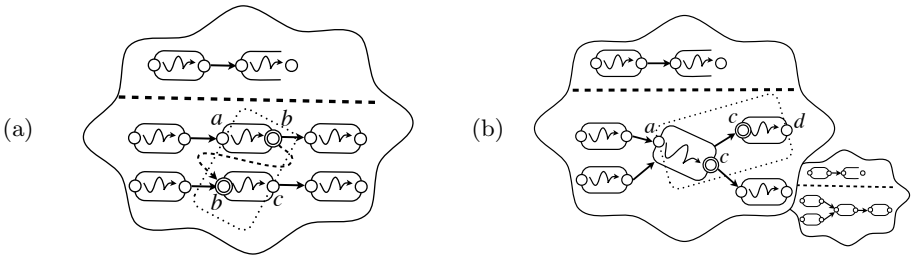


Fig. 6. The bag simplification operations: (a) merging two disconnected but contiguous round summaries $\langle a, b\rangle$ and $\langle b, c\rangle$ (resulting in (b)), and (b) merging two consecutive and contiguous summaries $\langle a, c\rangle$ and $\langle c, d\rangle$ (resulting in the small adjacent bag).

Theorem 2. *The sequence of bounded compositional semantics forms a monotonically increasing chain whose limit is identical to the compositional semantics, i.e.,* $\{\!|P|\!\}^0_{\mathrm{c}} \subseteq \{\!|P|\!\}^1_{\mathrm{c}} \subseteq \ldots \subseteq \bigcup_{k\in\mathbb{N}} \{\!|P|\!\}^k_{\mathrm{c}} = \{\!|P|\!\}_{\mathrm{c}}$.

Remark For simplicity, we present a compositional semantics in which information (i.e., the summary bags) only flows up from each sub-task to the task that posted it. However, an even more compact semantics—in the sense that more behaviors are expressed with the same bag-size bound—is possible when each task not only returns summaries, but also *receives* summaries as a parameter. For example, to interleave $2k$ summaries of two sub-tasks, one can pass the k summaries of the first sub-task to the second, and merge them with the second's summaries, as they are created, keeping only k at a time. Otherwise, one must interleave the two tasks' summaries outside, which means keeping $2k$ summaries for the enclosing task to interleave. Since the extension is purely technical, and not so insightful, we omit its description here. However, the results stated in the remainder of this paper are presented in terms of the bag-size bound w.r.t. this extension.

Note on Complexity For the case where variables have finite-data domains, determining whether a global valuation is reachable is NP-complete[6] (in k) for the bounded-task k-context bounded and unbounded-task k-delay bounded sequentializations [19, 7], and PSPACE-complete (in k) for the unbounded-task k-context bounded sequentialization [16]. Since these cases are reducible to our semantics (see Section 5), it follows that global-state reachability is PSPACE-hard (in k) in the most general instance of our framework. Since the number of bag configurations is (singly) exponential in k, membership in EXPTIME is not hard to obtain. The practical difference between the NP/PSPACE-complete complexities of other sequentialization-based analyses is unclear; as sub-exponential time algorithms are not known for NP problems, exponential time is spent in the

[6] Here the literature has assumed a fixed number of finite-state program variables in order to ignore the effect of complexity resulting from *data*. Indeed the reachability problem is EXPTIME-complete in the number of variables, due to the logarithmic encoding of states in the corresponding pushdown system.

worst case, either way. Note however, that these complexity considerations are of limited significance, since we target an arbitrary class of sequential programs—not only programs with finite-data.

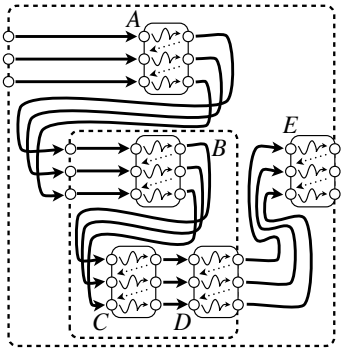
5 Global-Round Explorations

To understand the relationship between our bounded compositional exploration and the existing sequentializable analyses, we proceed in two steps. First we describe a restriction of our bounded semantics in which every task agrees on a global notion of execution "rounds," i.e., where each task executes (at most) one uninterrupted segment per global round. Second we show that this restriction captures and unifies the existing sequentializable analyses based on bounded context-switch and bounded delay.

A k *global-round execution* of a program P is an asynchronous execution of P where (i) each task executes in (up to) k uninterrupted segments called "rounds"—with the restriction that sub-tasks can only execute in, or after, the round in which it is created—and (ii) the i^{th} round of every task is executed before the $(i+1)^{\text{st}}$ round of *any* task, in the depth-first order over the task-creation tree; see Fig. 7a. Thus we can view each task as executing across a grid of k rounds, being interrupted $k-1$ times by the other tasks. With this view, each task can be characterized by an *interface* consisting of $2k$ global state valuations: the k global-state valuations seen by the task at the beginning of each round, and the k global-state valuations seen at the end (as in Fig. 7a). A task's interfaces are computed by guessing the initial global-state valuation of each round, following some number of sequential transitions, then nondeterministically advancing to the next round, keeping the computed local state, but updating the global-state to the next-guessed valuation. Given the interfaces for each task, sequentialization of the k global-round schedules is possible, by a reduction that executes each task once, according to the linear task-order, over a k-length vector of global-state valuations. The k *global-round semantics* of P, written $\{\!|P|\!\}^k_{\text{gr}}$ is the set of global valuations reachable in a k global-round execution of P.

Though we have defined the global-round semantics w.r.t. the asynchronous semantics, in fact we can restrict the k-bounded compositional semantics to compute only k global-round interfaces. During construction of the j^{th} round-summary of the current task t, the export-list contains $j-1$ summaries (of rounds $1 \ldots j-1$) for t and its current sub-tasks, i.e., all descendants of t's thus-far posted tasks. At the same time, the bag maintains a $(k-j+1)$-length list of summaries accumulating the rounds $j \ldots k$ of t's current sub-tasks. Just before t begins round $j+1$, the accumulated summary of round j for t's current sub-tasks is consumed, so that the exported summary of round j captures the effect of one round of t, followed by one round of t's current sub-tasks. When t posts another task t_i (in round j), the $k-j+1$ summaries exported by t_i—note t_i and its descendants can only execute after round j of t—are combined with the $k-j+1$

(a)



(b)

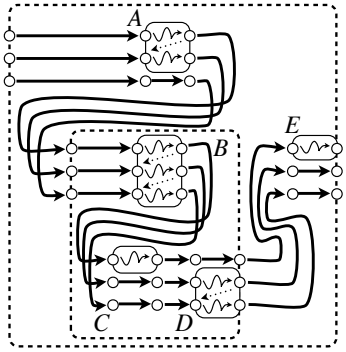


Fig. 7. (a) 3 global-round exploration, and (b) 4-delay exploration, for a program in which task A creates B and E, and task B creates C and D. Each task's interface (drawn with dashed-lines) summarizes 3 uninterrupted computations (drawn with squiggly arrows) of the task and all of its sub-tasks. Bold arrows connect global-state valuations, and dotted arrows connect local-state valuations. Note that the 3-round exploration allows each task 2 interruptions, while the 4-delay exploration allows all tasks combined only 4 interruptions.

accumulated summaries of t's current sub-tasks.[7] Thus when t completes round k, each round i summarizes round i of t followed by round i of each sub-task of t, in depth-first order, over the ordered task-creation tree of t.

Theorem 3. *The k global-round semantics is subsumed by the k-bounded compositional semantics, i.e., for all programs P we have $\{\!|P|\!\}^k_{\text{gr}} \subsetneq \{\!|P|\!\}^k_{\text{c}}$.*

In fact, our compositional semantics captures many more behaviors than k global-round analyses. For example, many behaviors cannot be expressed in k (global) rounds, though can be expressed by decoupling the rounds of different tasks. Intuitively, the additional power of our compositional semantics is in the ability to *reuse* the given bounded resources *locally*, ensuring only that the number of *visible* resources at any point is bounded. Appendix D illustrates an example demonstrating this added analysis power.

Theorem 4. *There exists a program P, $k_0 \in \mathbb{N}$, and a sequence $g_1, g_2, \ldots \in \{\!|P|\!\}^{k_0}_{\text{c}}$ of global-state valuations such that for all $k \in \mathbb{N}$, $g_k \notin \{\!|P|\!\}^k_{\text{gr}}$.*

5.1 Context-bounding

In its initial formulation, the so-called "context-bounded" (CB) analyses explored the asynchronous executions of a program with a fixed number of statically allocated tasks, with at most two *context-switches* between tasks [22]. Shortly

[7] Using the extension described at the end of Section 4, this combination does not require additional bag space.

thereafter, Qadeer and Rehof [21] extended CB to explore an arbitrary bound k of context-switches.

Later, Lal and Reps [19] proposed a linear "round-robin" task-exploration order, and instead of bounding the number of context-switches directly, bounded the number of rounds in an explored round-robin schedule between n tasks. (It is easy to see that every k-context bounded execution with an unrestricted scheduler is also a k-round execution with a round-robin scheduler.) With this scheduling restriction, it becomes possible to summarize each task i's execution by an interface of k global-state valuation pairs, describing a computation that is interrupted by tasks $(i+1), (i+2), \ldots, n, 1, 2, \ldots, (i-1)$, $k-1$ times. In fact, this schedule is just a special case of the k global-round exploration, restricted to programs with a fixed number of statically-created tasks. La Torre et al. [16]'s subsequent extension of this k-round round-robin exploration to programs with a *parameterized* amount of statically-created tasks is also a special case of k global-round exploration, restricted to programs with an *arbitrary* number of statically-created tasks.

To formalize this connection, let a *static-task program* be an asynchronous program P which does not contain **post** statements, and an initial configuration $\langle g, \langle \ell, s \rangle, \emptyset \rangle$ is *(resp., parameterized) static-task initial* when s is of the form

$$\textbf{post}\ p_1\ \texttt{();}\ \ldots\texttt{;}\ \textbf{post}\ p_n\ \texttt{()} \qquad (\text{resp., } \textbf{while}\ \star\ \textbf{do}\ \textbf{post}\ p\ \texttt{()}).$$

A *k-round (resp., parameterized) CB execution* of a static-task program P is an asynchronous execution of P from a (resp., parameterized) static-task initial configuration c_0, where the initially posted tasks are dispatched in a round robin fashion over k rounds. The *k-round (resp., parameterized) CB semantics* of P, written $\{\!|P|\!\}^k_{\mathrm{cb}}$ (resp., $\{\!|P|\!\}^k_{\mathrm{cb}^*}$) is defined, as before, as the set of global valuations reachable from c_0 in a k-round (resp., parameterized) CB execution of P.

Theorem 5. *The k-round (parameterized) CB semantics is equal to the k global-round semantics, i.e., for all static-task programs P we have $\{\!|P|\!\}^k_{\mathrm{cb}^{(*)}} = \{\!|P|\!\}^k_{\mathrm{gr}}$.*

5.2 Delay-bounding

Emmi et al. [7]'s recently introduced delay-bounded (DB) exploration[8] expands on the capabilities of Lal and Reps [19]'s k-round CB exploration with its ability to analyze programs with dynamic task-creation (i.e., with arbitrary use of the **post** statement). Like CB, the DB exploration considers round-based executions with a linear task-exploration order, though with dynamic task-creation the order must be defined over the task-creation tree; DB traverses the tree depth-first.

In fact, Emmi et al. [7]'s delay-bounded semantics is a variation of the k global-round semantics which, for the same bound k, expresses many fewer asynchronous executions. In essence, instead of allowing each task $k-1$ interruptions, the budget of interruptions is bounded globally, over the entire asynchronous execution; see

[8] Since we are interested here in analyses amenable to sequentialization, we consider only the depth-first delay-bounded task-scheduler [7].

Fig. 7b. It follows immediately that each task executes across at most k rounds, in the same sense as in the k global-round semantics. Since the mechanism behind delay-bounding is not crucial to our presentation here, we refer the reader to Emmi et al. [7]'s original work for the precise definition of *k-delay executions*. The *k-delay semantics* of P, written $\{\!|P|\!\}^k_{\text{db}}$ is the set of global valuations reachable in a k-delay execution of P.

Theorem 6. *The k-delay semantics is subsumed by the k global-round semantics, i.e., for all programs P we have $\{\!|P|\!\}^k_{\text{db}} \subseteq \{\!|P|\!\}^k_{\text{gr}}$.*

However, like the separation between k-global round semantics and k bounded semantics, there are families of behaviors that can be expressed with a fixed number of global rounds, though cannot be expressed with a fixed number of delays: for instance, behaviors which requires an unbounded number of tasks be interrupted (once) cannot be expressed with any finite number of delays.

Theorem 7. *There exists a program P, $k_0 \in \mathbb{N}$, and a sequence $g_1, g_2, \ldots \in \{\!|P|\!\}^{k_0}_{\text{gr}}$ of global-state valuations such that for all $k \in \mathbb{N}$, $g_k \notin \{\!|P|\!\}^k_{\text{db}}$.*

5.3 Context-bounding vs. Delay-bounding

It follows from Theorems 5 and 6 that context-bounding simulates delay-bounding on static-task programs. In fact, we can also show that delay-bounding simulates context-bounding, for programs with a fixed number of tasks, by combining Theorem 5 with the following theorem.

Theorem 8. *For a fixed number n of tasks, the k global-round semantics is subsumed by the nk-delay semantics, i.e., for all static-task programs P with n-tasks we have $\{\!|P|\!\}^k_{\text{gr}} \subseteq \{\!|P|\!\}^{nk}_{\text{db}}$.*

However, by Theorems 5 and 7, delay-bounding cannot simulate k-round parameterized context-bounded executions, since no fixed number of delays can express the unbounded number of potential context-switches.

6 Related Work

The technique of reducing a concurrent program behaviors to sequential program behaviors has garnered much attention in recent years. Based on the notion of *context-bounding* [22, 21, 20], Lal and Reps [19] showed how to encode the bounded-round round-robin schedules of a concurrent program with statically-allocated tasks as a sequential program. La Torre et al. [14] gave a more efficient encoding—in the sense that unreachable global-state valuations are never explored—and later extended the approach to programs with an unbounded number of statically-allocated tasks [16]. Emmi et al. [7] have recently extended the basic insight of round-based scheduling to sequentialize programs with an unbounded number of dynamically-created tasks. Empirical evidence suggests

such techniques are indeed useful for bug-finding [20, 18, 10, 16]. For a more thorough comparison of these sequentializations, see Section 5.

Recently Kidd et al. [13] have shown how to sequentialize priority preemptive scheduled programs, and Garg and Madhusudan [9] have proposed an overapproximating sequentialization, albeit by exposing task-local state to other tasks. Both reductions assume a finite number of statically-declared tasks.

More generally, sequentialization could be seen as any linear traversal of the task-creation tree. The sequentializations we consider here are restricted to depth-first traversal, since they target sequential recursive programs, whose unbounded structure is, in general, contained to the procedure stack; the stack-based data-structure used for the depth-first traversal can be combined with the program's procedure stack. However, if one is willing to target other program models, one can consider other task-tree traversals, e.g., breadth-first using queues, or a completely non-deterministic traversal respecting the task-creation order, keeping, for instance, a multiset of horizon tasks. Atig et al. [1, 2]'s bounded explorations of programs with dynamic task-creation, by reduction to Petri-net reachability, are sequentializable in this sense.

Our characterization of sequentializability could also be relaxed to allow the exchange of local-state valuations (or alternatively, unbounded sequences of global-state valuations) between tasks. For instance, explorations based on *bounded languages* [12, 8] take this approach, essentially restricting concurrent exploration to inter-task interactions conforming to a regular *pattern*; then each task is executed in isolation by taking its product with the pattern-automaton. We simply note that the existing sequentializations avoid the (possibly expensive) computation of such a product.

7 Conclusion

We have proposed a framework for parameterized and compositional concurrent program analysis based on reduction to sequential program analysis. Our framework applies to a general class of shared-memory concurrent programs, with arbitrary preemption and dynamic task-creation, and strictly captures the known (round-based) sequentializations. It is our hope that this understanding will lead to further advances in sequential reductions, e.g., by enlightening efficiently-encodable instances of the general framework.

Though we have unified the existing sequentializations while maintaining their desirable qualities (i.e., sequential program model, compositionality, parameterization) by relaxing the global round-robin schedule, we are aware of one remaining restriction imposed by our framework. Besides the round-robin restriction imposed by the existing sequentializations, there is an implicit restriction imposed by translating task-creation directly to procedure calls: the tasks created by a single task are visited in the order they are created. Though further generalization is possible (e.g., by adding unbounded counters to the target program), such reductions will likely lead to much more complex analyses.

Bibliography

[1] 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.

[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 and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM, 2002.

[4] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 159–169. ACM, 2008.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[6] 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.

[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] P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. In *CAV '10: Proc. 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 600–614. Springer, 2010.

[9] P. Garg and P. Madhusudan. Compositionality entails sequentializability. In *TACAS '11: Proc. 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer, 2011.

[10] 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.

[11] B. Jannet and A. Miné. The Interproc analyzer. `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`.

[12] V. Kahlon. Tractable dataflow analysis for concurrent programs via bounded languages, July 2009. Patent WO/2009/094439.

[13] 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.

[14] 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.

[15] S. La Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI '09: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222. ACM, 2009.

[16] 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.

[17] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–182. ACM, 2008.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] T. W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

A Syntactic Sugar

The following syntactic extensions are reducible to the original syntax of asynchronous programs of Section 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 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 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 of Theorems

Theorem 1. *The compositional semantics and asynchronous semantics are identical, i.e., for all programs P we have $\{\!|P|\!\}_{\mathrm{c}} = \{\!|P|\!\}_{\mathrm{a}}$.*

Proof (Sketch). To show $\{\!|P|\!\}_{\mathrm{a}} \subseteq \{\!|P|\!\}_{\mathrm{c}}$, let h be an asynchronous execution with tasks uniquely identified by $\mathsf{TaskIDs}$, and define $\mathsf{Segs}_h = \mathsf{TaskIDs} \times \mathsf{Vals} \times \mathsf{Vals} \times \wp(\mathsf{TaskIDs})$ such that $\langle u, g, g', U\rangle \in \mathsf{Segs}_h$ if and only if task u has an uninterrupted execution from g to g' in h posting tasks U. Then define a *segment causality relation* $< \subseteq \mathsf{Segs}_h \times \mathsf{Segs}_h$ as the smallest transitively-closed relation such that for two segments $s_1 = \langle u_1, g_1, g_1', U_1\rangle \in \mathsf{Segs}_h$ and $s_2 = \langle u_2, g_2, g_2', U_2\rangle \in \mathsf{Segs}_h$, $s_1 < s_2$ if either (a) $u_1 = u_2$ and s_1 appears before s_2 in h, or (b) $u_2 \in U_1$.

We show by induction on the sub-task relation how to construct a complete task-segment sequence σ_{u_0} corresponding to the execution h starting from an initial task u_0. For $u \in \mathsf{TaskIDs}$, we define h_u as the sub-sequence of h containing only configurations of u and u's (transitive) sub-tasks. As the basis, when u_0 does not post any additional tasks, then the total order on segments of task u_0 defined by h_{u_0} is easily replicated in the compositional semantics, using the NextRound rule into a segment sequence σ_{u_0}.

Given two words $\sigma, \tau \in \Sigma^*$ of some alphabet Σ, the *shuffle* of σ and τ, denoted $\sigma \bowtie \tau$, is the set of words $\sigma_1\tau_1\sigma_2\tau_2 \ldots \in \Sigma^*$ such that $\sigma = \sigma_1\sigma_2\ldots$ and $\tau = \tau_1\tau_2\ldots$. The *shuffle* of a sequence of words is obviously defined, given that $\bowtie$ is commutative and associative.

When u_0 posts tasks $u_1, \ldots, u_i$, the inductive hypothesis defines for each h_{u_j} a segment sequence σ_{u_j}. As h_{u_0} is given by some shuffling of $h_{u_1}, \ldots, h_{u_i}$ with the segments of task u_0 (respecting the segment causality relation $<$), we must construct a segment sequence σ_{u_0} given by an arbitrary shuffling of $\sigma_{u_1}, \ldots, \sigma_{u_i}$—given by the SubTask rule—with the segments of task u_0 (again, respecting the segment causality relation $<$). It is not hard to verify that using the Interleave, NextRound, and Resume rules, the compositional semantics does indeed consider every such shuffling, sequencing each segment one at a time into an exported total order; for each segment taken from a sub-task (given by the SubTask rule), the compositional execution begins a new round (via NextRound), and consumes the given segment (via Interleave); for each segment of the current task, the compositional execution simply begins a new round, and applies a sequence of sequential rules. The causality relation $<$ is never violated, since a sub-task's segments cannot be sequenced until they are provided by a corresponding SubTask rule, and segments of the same task are sequenced in their causal order.

In the initial task u_0, the compositional execution behaves similarly, except does not apply the NextRound rule. Since h is a valid asynchronous execution, the segments of σ_{u_0} are contiguous—i.e., the global valuation reached by each segment is equal to the initial global valuation of the next. Thus the Interleave rule can be repeatedly applied, stitching together the contiguous segments, building a complete compositional execution from σ_{u_0}.

Reasoning in the other direction (i.e., $\{\!|P|\!\}_{\mathrm{c}} \subseteq \{\!|P|\!\}_{\mathrm{a}}$) proceeds similarly, essentially showing that any successfully-stitched (via INTERLEAVE) segment sequence σ without using the NEWROUND rule corresponds to an asynchronous execution of the contiguous segments of σ. □

Theorem 2. *The sequence of bounded compositional semantics forms a monotonically increasing chain whose limit is identical to the compositional semantics, i.e.,* $\{\!|P|\!\}^0_{\mathrm{c}} \subseteq \{\!|P|\!\}^1_{\mathrm{c}} \subseteq \ldots \subseteq \bigcup_{k\in\mathbb{N}} \{\!|P|\!\}^k_{\mathrm{c}} = \{\!|P|\!\}_{\mathrm{c}}$.

Proof. This is immediate, since every k-bounded compositional execution is both a $(k+1)$-bounded execution, and a compositional execution. Furthermore, every compositional execution uses only finite-sized bags.

C The Sequential Program Statements

For a statement context S (see Section 2), we write $\langle g, \langle \ell, S\rangle\, w, m\rangle\, [s]$ to denote the configuration $\langle g, \langle \ell, S[s]\rangle\, w, m\rangle$. For convenience we also define

$$
\begin{aligned}
e(\langle g, \langle \ell, s,\rangle\, w, m\rangle) &\stackrel{\text{def}}{=} e(g, \ell) && \text{expression evaluation}\\
\langle g, w, m\rangle \cdot w' &\stackrel{\text{def}}{=} \langle g, w'w, m\rangle && \text{frame push/pop}\\
\langle g, w, m\rangle\, (\mathtt{g} \leftarrow g') &\stackrel{\text{def}}{=} \langle g', w, m\rangle && \text{global update}\\
\langle g, \langle \ell, s\rangle\, w, m\rangle\, (\mathtt{l} \leftarrow \ell') &\stackrel{\text{def}}{=} \langle g, \langle \ell', s\rangle\, w, m\rangle && \text{local update.}
\end{aligned}
$$

Fig. 8 gives the transition relation $\rightarrow^{\mathrm{s}}_P$ for the sequential program statements. 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 in code-to-code translations in the guess-and-constrain style sequential program reductions [19, 7].)

D Example: Beyond Global-round Analyses

Fig. 9a gives a program P in which every reachable state is reachable (with a fixed bound) in the bounded compositional semantics (by interleaving a `B` task at the yield point of each `A` task), though there is no bound such that global-round semantics express every reachable state. Note that any k global-round exploration only explores `y`-values less than k, since (i) every `A` task must be executed between two distinct rounds to increment `y`, and (ii) each posted `A` task must begin after the second round of the `A` task that posted it.

The compositional semantics explores every positive value $j \in \mathbb{N}$ of `y` with bag-sizes bounded by $k = 4$ (see Fig. 9b): each `A` task exports a pair of segment summaries—i.e., before the yield point, and after—and each `B` task exports a single summary. Each parent task (i.e., the one posting) saves one summary from the beginning of `A` (with $\mathtt{y} = i$) to the yield point (with $\mathtt{x} = \mathtt{false}$, $\mathtt{y} = i$), and

$$\textsc{Skip}\ \frac{}{c[\mathbf{skip}; s] \to^{\mathrm{s}}_P c[s]} \qquad \textsc{Assume}\ \frac{\texttt{true} \in e(c)}{c[\mathbf{assume}\ e] \to^{\mathrm{s}}_P c[\mathbf{skip}]}$$

$$\textsc{If-Then}\ \frac{\texttt{true} \in e(c)}{c[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to^{\mathrm{s}}_P c[s_1]} \qquad \textsc{If-Else}\ \frac{\texttt{false} \in e(c)}{c[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to^{\mathrm{s}}_P c[s_2]}$$

$$\textsc{Loop-Do}\ \frac{\texttt{true} \in e(c)}{c[\mathbf{while}\ e\ \mathbf{do}\ s] \to^{\mathrm{s}}_P c[s; \mathbf{while}\ e\ \mathbf{do}\ s]} \qquad \textsc{Loop-End}\ \frac{\texttt{false} \in e(c)}{c[\mathbf{while}\ e\ \mathbf{do}\ s] \to^{\mathrm{s}}_P c[\mathbf{skip}]}$$

$$\textsc{Assign-Global}\ \frac{v \in e(c)}{c[\mathtt{g} := e] \to^{\mathrm{s}}_P c[\mathbf{skip}](\mathtt{g} \leftarrow v)} \qquad \textsc{Assign-Local}\ \frac{v \in e(c)}{c[\mathtt{l} := e] \to^{\mathrm{s}}_P c[\mathbf{skip}](\mathtt{l} \leftarrow v)}$$

$$\textsc{Call}\ \frac{v \in e(c)}{c[\mathbf{call}\ x := p\ e] \to^{\mathrm{s}}_P c[x := \star] \cdot \langle v, s_p \rangle} \qquad \textsc{Return}\ \frac{v \in e(c \cdot \langle \ell, S[\mathbf{skip}] \rangle)}{c[x := \star] \cdot \langle \ell, S[\mathbf{return}\ e] \rangle \to^{\mathrm{s}}_P c[x := v]}$$

Fig. 8. The transition relation $\to^{\mathrm{s}}_P$ for the sequential statements of a program P.

begins a second segment from the global value $\mathtt{x}$= **true**, $\mathtt{y} = i$. In the second segment we then take the **then** branch, increment $\mathtt{y}$, and import bags from the posted `A` and `B` tasks. From the incremented value of $\mathtt{y}$ ($i + 1$ in Fig. 9b) we apply the first segment of the posted `A` task, followed by the only segment of `B`, followed by the second `A` segment, ending with $\mathtt{y} = j$. In this way, each `A` task is interrupted by a `B` task, and is allowed to increment $\mathtt{y}$. As the number of recursive posts to `A` is never bounded, the value of $\mathtt{y}$ is incremented an arbitrary number of times.

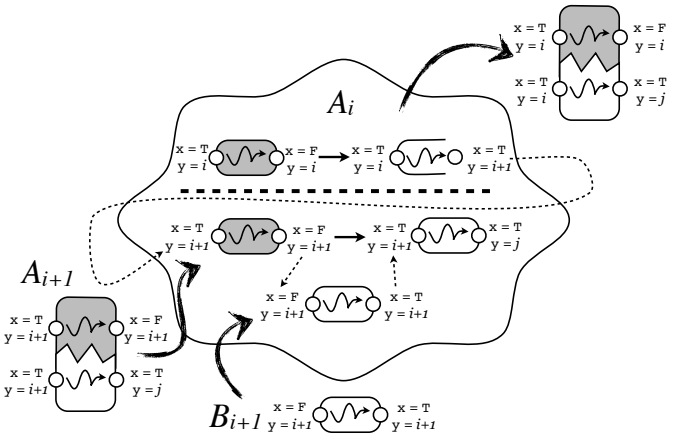
```
var x:bool, y:int

proc A()
  x := false;
  yield;
  if x then
    y := y + 1;
    post A;
    post B

proc B()
  x := true

init: post A ();
      post B ()
```

(a)



(b)

Fig. 9. A 4-bounded compositional exploration of a program with an unbounded number of nested sub-tasks. We abbreviate `true` and `false` with `T` and `F`.