Exploiting Synchronization for the Static Detection of Programming Errors

Michael Emmi
IMDEA Software Institute
michael.emmi@imdea.org

Burcu Külahçıoğlu Özkan
Koç University
bkulahcioglu@ku.edu.tr

Serdar Tasiran
Koç University
stasiran@ku.edu.tr

ABSTRACT

As asynchronous programming becomes more mainstream, program analyses capable of automatically uncovering programming errors are increasingly in demand. Since asynchronous program analysis is computationally costly, current approaches sacrifice completeness and focus on a limited set of thread schedules empirically likely to expose programming errors. These approaches make use of a *parameterized* thread scheduler that explores a set of schedules similar to a default deterministic schedule. By increasing the parameter, a larger set of thread interactions can be explored but at a higher cost. The effectiveness of these approaches depends critically on the default (deterministic) scheduler on which varying schedules are fashioned.

We find that the limited exploration of relevant asynchronous program behaviors can be made more efficient by designing parameterized schedulers that correspond well with the intended synchronization and ordering of program events, e.g. arising from waiting for an asynchronous task to complete. We follow a reduction-based "sequentialization" approach to analyzing asynchronous programs that can leverage existing (sequential) program analysis tools by encoding the program executions according to a "synchronization-aware" scheduler as executions of a sequential program. Analysis based on our new scheduler comes at no greater computational cost, and provides strictly greater behavioral coverage than analysis based on existing parameterized schedulers; we validate these claims both conceptually, with complexity and behavioral-inclusion arguments, and empirically, by discovering actual reported bugs faster with smaller parameter values.

1. INTRODUCTION

In order to improve program performance and responsiveness, many modern programming languages and libraries promote an asynchronous programming model, in which "asynchronous procedures" can execute concurrently with their callers, until their callers explicitly wait for their completion. Accordingly, as concurrently-executing procedures interleave their accesses to shared memory, asynchronous programs are prone to concurrency-related errors.

In this work, we develop program analyses capable of detecting errors in asynchronous programs. To motivate the need for such analyses, consider the subtle error in the event-handling C# code of a graphical user interface found on StackOverflow, which is listed Figure 1. The `MySubClass.onNavigatedTo` method accesses image-related information (i.e. `m_bmp.PixelWidth`) which is filled in by the `LoadState` method, invoked by the `OnNavigatedTo` method of the base class. However, the `LoadState` method has been implemented to execute asynchronously so that its callers can continue to execute while the image file is read — which is presumably a high-latency operation — meaning that **`base`**`.OnNavigatedTo` can return before `m_bmp` has been initialized. This creates a race between the initialization of `m_bmp` and its use in the call to `Canvas.SetLeft`, which results in an error when its use wins. Not having anticipated this race, the programmer has failed to provide adequate synchronization to ensure that the call to `LoadState` completes before `m_bmp` is accessed by `OnNavigatedTo`.

While detecting such concurrency bugs by *exhaustive* exploration of all possible program schedules is intractable, one promising approach is the *prioritized* exploration of behaviors whose manifestations rely on a small numbering of ordering dependencies between program operations. In particular, the delay bounding approach [1] explores the program behaviors arising in executions with a given scheduler $S(K)$ parameterized by a "delay bound" $K \in \mathbb{N}$; while $S(0)$ is a deterministic scheduler, exhibiting only one order of program operations, $S(K)$ is given additional nondeterministic choice with each increasing value of K, allowing additional orders, and ultimately, exhibiting additional observable program behaviors. The approach is particularly compelling under the hypothesis that interesting program behaviors (e.g., bugs) manifest with few ordering dependencies: Emmi et al. [1] demonstrate an efficiently-implementable "depth-first" delaying scheduler $\mathrm{DF}(K)$ which can expose behaviors with few ordering dependencies using small values of K.

In practice, the cost of prioritized exploration with a parameterized scheduler $S(K)$ is highly sensitive to the value of K, limiting $\mathrm{DF}(K)$-based exploration to roughly $0 \leq K < 5$, depending on program size. While such small values of K may suffice to expose bugs in programs which use very little synchronization, each program synchronization statement induces another event-order dependency, possibly forcing

```
// MySubClass
BitmapImage m_bmp;
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  await PlayIntroSoundAsync();
  image1.Source = m_bmp;
  Canvas.SetLeft(image1, Window.Current.Bounds.Width - m_bmp.PixelWidth);
}
protected override async void LoadState(Object nav, Dictionary<String, Object> pageState)
{
  m_bmp = new BitmapImage();
  var file = await StorageFile.GetFileFromApplicationUriAsync("ms-appx:///pic.png");
  using (var stream = await file.OpenReadAsync())
    await m_bmp.SetSourceAsync(stream);
}
// base class
class LayoutAwarePage : Page
{
  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
              // ...
    this.LoadState(e.Parameter, null);
  }
}
```

Figure 1: This code contains a subtle bug due to a race condition on the `m_bmp` field.

DF(K) to further deviate from its natural deterministic order by increasing K. For instance, if DF(K)'s default schedule encounters a statement which acquires a lock held by another thread, then DF(K) must spend one of its K delays in order to execute the other thread and eventually progress past the lock acquisition. In the context of asynchronous programs, e.g., using C#'s asynchronous methods, DF(K) must spend one if its K delays to advance past a statement which waits for a non-completed task to complete. It follows that program behaviors which can appear only after a high number of synchronization statements carry a high number of event-order dependencies, which ultimately may be exercised by DF(K) only for large values of K. As the cost of program exploration with DF(K) is sensitive to K, the discovery of such behaviors may require an unreasonable amount of computing resources.

In this work we demonstrate a delaying scheduler DFW(K) for which the cost of exploration is not tied to program synchronization, and yet which still enjoys DF(K)'s strengths, in particular:

- **Sequentialization** The program executions allowed by DFW(K) can be simulated by a sequential program with nondeterministically-chosen data values.
- **Low Complexity** The reachability problem finite-data programs restricted to DFW(K) executions is NP-complete[1] in K.

However, unlike DF(K), the DFW(K) scheduler explicitly takes program synchronization into account in its scheduling decisions, so that event-order dependencies arising from synchronization statements do not force K to increase. Effectively, this means that DFW(K) provides strictly greater behavioral coverage than DF(K) at virtually no additional cost.

[1]This complexity assumes program variables are fixed in number and size.

Our contributions and outline are:

§2 A formal semantics of asynchronous programs with synchronization.

§3 A formal description of the DFW(K) scheduler, and comparison with DF(K).

§4 A "compositional semantics" for DFW(K), which fosters sequentialization.

§5 A code translation encoding DFW(K) executions as a sequential program.

§6 NP-completeness of reachability under DFW(K) for finite data programs.

§7 An empirical comparison: DFW(K) finds bugs faster, with smaller K.

In theory, every program behavior observable with DF(K) is also observable with DFW(K) for any K, yet the reverse is untrue: for any K_0 there exist programs whose sets of behaviors observable with DFW(K_0) are not all observable with DF(K) for any K; Section 3 demonstrates this fact. Empirically, Section 7 demonstrates that our sequentialization of DFW(K) is more effective than DF(K) in finding bugs in real code examples as the number of synchronization operations grows.

While our development is centered around a simple programming model with asynchronous procedure calls, and "wait" statements which block until the completion of a given asynchronous call, our technical innovations also apply to other asynchronous programming primitives provided by widely-used programming languages, such as the partially-synchronous procedure calls of C#[2] and the wait-for-all

[2]In C#, executing an "await" inside of a procedure returns control to the caller, executing the remaining continuation asynchronously.

synchronization barriers of, e.g., Cilk and X10. We believe that the same principles would also apply for other synchronization mechanisms such as semaphores and locks.

2. ASYNCHRONOUS PROGRAMS

In order to develop our theory around synchronization-aware schedulers, we introduce a formal model of asynchronous programs with asynchronously executing procedure calls, and blocking wait-for-completion synchronization. When a procedure is called asynchronously, control returns immediately to the caller, who may store a *task identifier* with which to refer to the procedure instance, which we henceforth refer to as a *task*. The newly-created task then executes concurrently with the caller, possibly accessing the same set of global program variables concurrently. While we suppose for simplicity that task identifiers are not stored in global program variables, we do allow task identifiers to be stored in procedure-local variables, passed as arguments to called procedures, and returned from procedure calls. A task identifier i may be used to block the execution of another task j until task i completes, at which point the task's result may be stored in a program variable. Together these features comprise an expressive model of concurrent programs which corresponds closely to the features in a diverse range of programming languages including C#, Cilk, and X10.

Syntacticly, a program is a set of global variable declarations, along with a set of procedure declarations whose statements are given by the grammar:

s ::= s ; s | **assume** e | **skip**
| **if** e **then** s **else** s | **while** e **do** s
| x := e | **call** x := p e | **return** e
| **async** x := p e | x := **wait** e

Here, x ranges over the set Vars of program variables, p ranges over procedure names, and e ranges over program expressions — whose grammar we leave unspecified. The set of program values Vals includes the set IDs of task identifiers, including a special polymorphic nil value $\perp$. We assume program expressions are statically typed, that each task-identifier typed expression evaluates to a single value $i \in \mathsf{IDs}$, and that each non-identifier typed expression evaluates to a set of values $V \subseteq (\mathsf{Vals} \setminus \mathsf{IDs})$. Furthermore, we suppose that the set of program expressions contains $\star$, which can evaluate to any non-identifier program value, and that each program contains a single entry-point procedure named main.

Aside from the usual sequential programming statements, we include the statement **async** x := p e which creates a new task to execute procedure p with argument e, storing its identifier in the procedure-local variable x, and the statement x := **wait** e, which blocks execution until the task $i \in \mathsf{IDs}$ referenced by e completes, at which point the result which i returns is assigned to the variable x. Furthermore, to facilitate our translations of programs into *sequential* programs with nondeterministically-chosen values, which appear in later sections, we include the **assume** e statement, which proceeds only if the expression e evaluates to **true**, and the nondeterministic assignment x := $\star$.

A *frame* $f = \langle \ell, s \rangle \in \mathsf{Frames}$ is a valuation $\ell : \mathsf{Vars} \to \mathsf{Vals}$ to procedure-local variables, along with a statement $s \in \mathsf{Stmts}$ describing the entire body of a procedure that remains to be executed; s_p denotes the statement body of procedure p. A *task* $t = \langle i, w, v \rangle$ is an identifier $i \in \mathsf{IDs}$, along with a procedure frame stack $w \in \mathsf{Frames}^*$, and a result value $v \in (\mathsf{Vals})$. We say a task $t = \langle i, w, v \rangle \in \mathsf{Tasks}$ is *completed* when $v \neq \perp$, and we maintain that $v = \perp$ if and only if $w = \varepsilon$[3]; we refer to t as the *root task* $i = \perp$. A *task pool* is a set $m \subseteq \mathsf{Tasks}$ in which no two tasks have the same identifier. A *configuration* $c = \langle g, m \rangle \in \mathsf{Configs}$ is a valuation $g : \mathsf{Vars} \to \mathsf{Vals}$ to the global program variables, along with a task pool m.

To reduce clutter in our definition of program semantics, we define a notion of contexts. A *statement context* S is a term derived from the grammar $S ::= \diamond | S; s$, 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. Intuitively, a context filled with s, e.g., $S[s]$, indicates that s is the next statement to execute in the statement sequence $S[s]$. Similarly, a task context $T = \langle l, S \rangle \cdot w$ is a frame sequence in which the first frame's statement is replaced with a statement context, and we write $T[s]$ to denote the frame sequence $\langle l, S[s] \rangle \cdot w$.

Figure 2 defines an operational program semantics via a set of transition rules on program configurations; Appendix A lists the remaining transition rules for the usual sequential program statements. The CALL rule invokes a procedure by adding a new procedure frame on top of the procedure frame stack. The ASYNC rule adds a newly-crated task to execute an asynchronously called procedure to the task pool, and stores its task task identifier (in a procedure-local variable). The CONTINUE rule progresses past a **wait** statement when the waited task is already completed, assigning its return value into the result variable. The COMPLETE rule completes a task which returns from its bottommost procedure frame, while the RETURN assigns the return value of a non-bottom procedure frame to the caller's result variable.

The *initial configuration* $c_0 = \langle g_0, m_0 \rangle$ of a program P is the valuation g_0 mapping each global variable of P to $\perp$, along with a task pool m_0 containing a single root task $\langle \perp, \langle \ell_0, s_{\mathsf{main}} \rangle, \perp \rangle$ such that ℓ_0 maps each variable of the main procedure to $\perp$. A *final configuration* $\langle g, m \rangle$ is a valuation g along with a task pool m in which all tasks are completed: for all $\langle \_, \_, v \rangle \in m$, $v \neq \perp$. An *execution* of a program P to c_j is a configuration sequence $\xi = c_0 c_1 \ldots c_j$ starting from the initial configuration c_0 such that $c_i \to c_{i+1}$ for $0 \leq i < j$; ξ is called *finalized* when c_j is final. We define $R(P)$ as the set of global valuations reached in finalized executions of P, i.e., $R(P) = \{ g : c_0 \to \ldots \to \langle g, \_ \rangle \text{ is finalized } \}$.

Our definition of the reachable valuations $R(P)$ is purposely restricted to final configurations due to our inclusion of nondeterministic expressions and the **assume** statement, which are needed by our sequentializations in the following sections. This definition of $R(P)$ does not lose any generality since any program can be transformed into one in which any configuration can reach a completed configuration with the same global valuation, e.g., by adding a **exit** flag to simulate the control flow of an uncaught program exception [2].

[3]We denote the empty sequence with ε.

$$\textsc{Call}\quad \frac{\ell \in e(g, M) \qquad f = \langle \ell, s_p \rangle}{\langle g, m \cup \{\langle i, T[\textbf{call } x := p\ e], v\rangle\}\rangle \to \langle g, m \cup \{\langle i, f \cdot T[x := \bot], v\rangle\}\rangle}$$

$$\textsc{Async}\quad \frac{\ell \in e(g, M) \qquad f = \langle \ell, s_p \rangle \qquad j \text{ is fresh}}{\langle g, m \cup \{\langle i, T[\textbf{async } x := p\ e], v\rangle\}\rangle \to \langle g, m \cup \{\langle i, T[x := j], v\rangle, \langle j, f, \bot\rangle\}\rangle}$$

$$\textsc{Continue}\quad \frac{j = e(g, T) \qquad \langle j, \_, v\rangle \in m \qquad v \neq \bot}{\langle g, m \cup \{\langle i, T[x := \textbf{wait } e], \bot\rangle\}\rangle \to \langle g, m \cup \{\langle i, T[x := v], \bot\rangle\}\rangle}$$

$$\textsc{Complete}\quad \frac{f = \langle \ell, S[\textbf{return } e]\rangle \qquad v \in e(g, \ell)}{\langle g, m \cup \{\langle i, f, \_\rangle\}\rangle \to \langle g, m \cup \{\langle i, \varepsilon, v\rangle\}\rangle}$$

$$\textsc{Return}\quad \frac{f = \langle \ell, S[\textbf{return } e]\rangle \qquad v \in e(g, \ell)}{\langle g, m \cup \{\langle i, f \cdot T[x := \_], \bot\rangle\}\rangle \to \langle g, m \cup \{\langle i, T[x := v], \bot\rangle\}\rangle}$$

Figure 2: Transition rules over program configurations.

3. THE DFW SCHEDULER

The asynchronous program semantics of the previous section are defined with respect to an implicit task *scheduler*, which enables any non-completed task to execute at any time. Computing the reachable global valuations $R(P)$ of arbitrary programs P is costly. One compelling approach for lowering the cost of program exploration is by considering specialized "delay bounded" schedulers with limited nondeterminism [1]. In this section, we provide a formal operational characterization of Emmi et al.'s K-delay bounded depth-first scheduler $\mathrm{DF}(K)$ [1], as well as our novel "synchronization-aware" variation $\mathrm{DFW}(K)$.

A *scheduler* $\Psi = \langle Q, q_0, \delta, \pi\rangle$ is a set Q of states with initial state $q_0 \in Q$, a transition function $\delta : Q \times ((\mathsf{IDs} \times \mathsf{Configs}^2) \cup \{\varepsilon\}) \to Q$, and a task-selection predicate $\pi : Q \to \mathsf{IDs}$. Intuitively, a scheduler state $q \in Q$ determines the task $\pi(q) \in \mathsf{IDs}$ which is enabled to make a transition. The non-deterministic scheduling choices made by schedulers are represented using a ε transition in which the states of tasks do not change, but the scheduler state may. We say the scheduler is *deterministic* when $\delta(q, \varepsilon) = q$ for all $q \in Q$. An Ψ-*execution* is an execution $c_0 c_1 \dots c_j$ such that there exists a sequence $q_0 q_2 \dots q_{j'} \in Q^*$ and a monotonic injection $f : j \to j'$ such that for each transition $c_i \to c_{i+1}$ of task u_i, for $0 \leq i < j$, we have $u_i \in \pi(q_{f(i)})$, and $\delta(q_{f(i)}, u_i, c_i, c_{i+1}) = q_{f(i)+1}$; additionally, $q_{i+1} = \delta(q_i, \varepsilon)$ for $0 \leq i < j'$ where $i \notin \mathrm{range}(f)$. We define $R(P, \Psi)$ as the set of global valuations reached in finalized Ψ-executions of P.

We define both the $\mathrm{DF}(K)$ and $\mathrm{DFW}(K)$ schedulers over states which represent the ordered tree of tasks of an execution, in which the children of each node i are the tasks which task i called, in the order in which they are called. Formally, the *Depth-First Scheduler* [1] is the scheduler $\mathrm{DF}(K) = \langle Q, q_0, \delta, \pi\rangle$ such that

- Q is the set of vertex-labeled trees $\langle V, E, \lambda, d\rangle$ with with vertices $V \subset \mathsf{IDs}$, edges E, and labeling function $\lambda : V \to (\{\mathsf{R}, \mathsf{C}\} \times \mathbb{N})$, assigning each vertex $\lambda(i) = \langle b, k\rangle$ a Ready or Completed status b and a round number $k \in \mathbb{N}$, along with a *delay counter* $d \in \mathbb{N}$.
- q_0 is the tree $\langle \{\bot\}, \emptyset, \{\bot \mapsto \langle \mathsf{R}, 0\rangle\}, 0\rangle$.

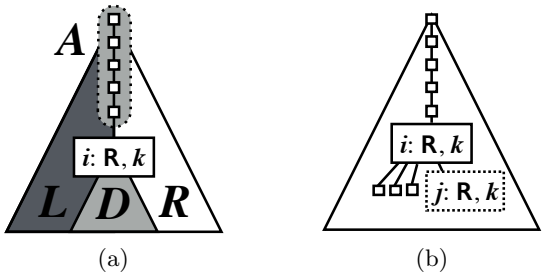

Figure 3: (a) A tree of the $\mathrm{DF}(K)$ scheduler enabling task i, showing i' ancestors (A), descendants (D), and the left (L) and right (R) descendants of i's ancestors. As i is enabled, each node in $A \cup L$ is either completed or has round $> k$, and each ndoe in $D \cup R$ is either completed or has round $\geq k$. (b) When task i posts j, $\mathrm{DF}(K)$ adds $(j \mapsto \mathsf{R}, k)$ as the rightmost child of i.

- $\pi(q)$ is the singleton set containing the least, in depth-first order, minimal-round ready vertex as in Figure 3(a), or $\emptyset$ when q does not contain such a vertex.
- $\delta(q, \varepsilon)$ is obtained from q by incrementing the delay counter d, and updating the label of vertex $\pi(q)$ from $\langle \mathsf{R}, k\rangle$ to $\langle \mathsf{R}, k+1\rangle$, so long as $d < K$; if $d \geq K$, then $\delta(q, \varepsilon) = q$.
- $\delta(q, i, c_1, c_2)$ is obtained from q by adding a rightmost child $(j \mapsto \langle \mathsf{R}, k\rangle)$ to vertex i, as in Figure 3(b), when $c_1 \to c_2$ is an Async transition creating task identifier j, and $\lambda(i) = \langle \mathsf{R}, k\rangle$.
- $\delta(q, i, c_1, c_2)$ is obtained from q by updating the label of vertex i from $\langle \mathsf{R}, k\rangle$ to $\langle \mathsf{C}, k\rangle$ when $c_1 \to c_2$ is a Complete transition.
- $d(q, i, c_1, c_2) = q$ for any other transition $c_1 \to c_2$.

Note that at each step of δ, the label of at most one task can change. Furthermore, $\mathrm{DF}(0)$ is deterministic.

Intuitively, $\mathrm{DF}(K)$ keeps track of a notion of execution *rounds* from $0 \dots K$ over which tasks execute, and executes lowest-

```
var i: int;

proc p()
  return i

proc main()
  var x: task
  var y: int

  i := 0;
  while ⋆ do
    async x := p();
    y := wait x;
    i := i + 1

  return
```

Figure 4: A program whose valuations are all reachable in DFW(0), yet are not all reachable in DF(K), for any $K \in \mathbb{N}$.

round tasks in depth-first order according to the task tree. For instance, DF(0) allows only a single round of execution, and executes each task in depth-first order until either all tasks are completed, or the task currently enabled by DF(0) blocks. DF(1) executes tasks according to the same order, except that the execution of one single task can increment the delay counter, and be postponed to the second round, resuming if and after which all other tasks complete in the first round. Note that when the currently enabled task in DF(K) is blocked, execution can only progress by advancing the blocked task to a subsequent round, and incrementing the delay counter. As the delay bound $K \in \mathbb{N}$ is increased, the cost of exploration can greatly increase, as DF(K) can allow exponentially more schedules.

To avoid increasing the delay bound $K \in \mathbb{N}$, which exponentially increases the number of alternate schedules explored, and ultimately increases the cost of exploration, we define a scheduler which does not enable tasks which are blocked waiting for others to complete. We define the *Synchronization-Aware Depth-First Scheduler* DFW(K) = $\langle Q, q_0, \delta, \pi\rangle$ by extending DF(K) with an additional waiting status label W, and defining $\delta(q, i, c_1, c_2)$ by applying the following post-processing function $f : (Q \times \mathsf{Configs}) \to Q$ to δ_{DF}: $\delta(q, i, c_1, c_2) = f(\delta_{\mathrm{DF}}(q, i, c_1, c_2), c_2)$, where $f(q, c)$ is obtained from q by

- updating the label of each $\langle \mathsf{R}, k\rangle$-labeled vertex i, such that i is waiting[4] for a $\langle \mathsf{R}/\mathsf{W}, \_\rangle$-labeled task j in c, to $\langle \mathsf{W}, k\rangle$.
- updating the label of each $\langle \mathsf{W}, \_\rangle$-labeled[5] vertex i, such that i is waiting for a $\langle \mathsf{C}, k\rangle$-labeled task j in c, to $\langle \mathsf{R}, k\rangle$.

Note that at each step of δ, the label of at most one task can change due to δ_{DF}, and the label of at most one task can change status to W due to f, though multiple labels can change status from W to R. Furthermore, DFW(0) is deterministic.

[4]We say i is *waiting for* j in $\langle g, m\rangle$ when $\langle i, T[x := \textbf{wait } e], \_\rangle \in m$ and $e(g, T) = j$.
[5]We additionally suppose that δ_{DF} is extended to treat W-labeled nodes as completed.

As the following result demonstrates, the DFW(K) scheduler is strictly more expressive than DF(K), in the sense that every global variable valuation which can be reached with DF(K) can also be reached with DFW(K), for all $K \in \mathbb{N}$, and that for every $K_0 \in \mathbb{N}$, there are programs whose set of valuations reached under DFW(K_0) cannot be reached by DF(K) for any finite value $K \in \mathbb{N}$; Figure 4 illustrates such a program, whose set of reachable valuations under DFW(0) is $\{\mathtt{i} \mapsto n : n \in \mathbb{N}\}$, while DF($K$) is restricted to $\{\mathtt{i} \mapsto n : n \leq K\}$, for any $K \in \mathbb{N}$. While this example may appear artificial at first, web programs that chain asynchronous calls are, in fact, quite common. If the loop in Figure 4 were replaced with one that repeats M times, with $M < K$, under the DF(K) scheduler, it would not be possible to complete program execution at all, since it would not be possible to move past the K-th iteration.

THEOREM 1. *$R(P, \mathrm{DF}(K)) \subseteq R(P, \mathrm{DFW}(K))$ for all programs P and $K \in \mathbb{N}$; for each $K_0 \in \mathbb{N}$ there are programs P for which $\bigcup_K R(P, \mathrm{DF}(K)) \subsetneq R(P, \mathrm{DFW}(K_0))$.*

4. COMPOSITIONAL SEMANTICS

Toward simulating the executions under our DFW(K) scheduler as the executions of a sequential program, we follow Bouajjani et al.'s intuition of *compositional* executions [3] with bounded task interfaces. Intuitively, a task interface is a summary of the effect on global storage of one task and all of its subtasks; literally, an interface is a sequence of global valuation pairs, with each pair summarizing a sequence of execution steps of a task and its subtasks. Compositional executions with bounded-size interfaces generalizes various bounding strategies for limiting concurrent behaviors to facilitate efficient program analysis, including context bounding [4, 5] and delay bounding [1]. In what follows we specialize Bouajjani et al.'s notion of compositional execution in order to fix a tight correspondence with the executions permitted by our DFW(K) scheduler.

A (K+1) *round interface* is a map $I : (K{+}1) \to (\mathsf{Vars} \to \mathsf{Vals})^2$ from natural numbers $k \in \mathbb{N} : k \leq K$ to pairs $I(k) = \langle g, g'\rangle$ of global variable valuations; we write $I(k).\mathsf{in}$ to denote g, and $I(k).\mathsf{out}$ to denote g', and we say I is *fresh* when $I(k).\mathsf{in} = I(k).\mathsf{out}$, for $0 \leq k \leq K$. To compose interfaces, we define a partial composition operator $\oplus$ such that $I \oplus J$ is defined when $|I| = |J|$ and $I(k).\mathsf{out} = J(k).\mathsf{in}$ for all $0 \leq k < |I|$, in which case $|I \oplus J| = |I|$ and $(I \oplus J)(k) = \langle I(k).\mathsf{in}, J(k).\mathsf{out}\rangle$ for all $0 \leq k < |I \oplus J|$. Furthermore, we say an interface I is *complete* when $I(k).\mathsf{out} = I(k+1).\mathsf{in}$ for $0 \leq k < |I| - 1$.

A *compositional configuration* $c = \langle g, w, k, d, I, J\rangle$ is a global valuation $g : \mathsf{Vars} \to \mathsf{Vals}$, along with a frame sequence $w \in \mathsf{Frames}^*$, a round index $k \in \mathbb{N}$, delay counter $d \in \mathbb{N}$, and interfaces I and J. Figure 5 defines a transition relation $\to$ on compositional configurations, and ultimately an interface generation relation $\rightsquigarrow$: the relation $\langle p, v_1, k_1\rangle \rightsquigarrow \langle I, d, v_2, k_2\rangle$ indicates that procedure p called with argument v_1 in round k_1, can return the value v_2. Furthermore, the effect of executing p and all of its subtasks, which executed up until round k_2 having spent d delays, is summarized by the interface I.

Intuitively, rather than adding a task to the pool, like the

$$\textsc{CAsync}\quad \frac{v_1 \in e(g,T) \qquad \langle p, v_1, k_1\rangle \rightsquigarrow \langle J_2, d_2, v_2, k_2\rangle \qquad d_1 + d_2 \leq K}{\langle g, T[\textbf{async } x := p\ e], k_1, d_1, I, J_1\rangle \rightarrow \langle g, T[x := \langle v_2, k_2\rangle], k_1, d_1 + d_2, I, J_1 \oplus J_2\rangle}$$

$$\textsc{CWait}\quad \frac{\langle v, k_2\rangle = e(g_1, T) \qquad g_1 = I[k_1].\mathsf{out} \qquad g_2 = J[k_2].\mathsf{out} \qquad I[k].\mathsf{in} = I[k].\mathsf{out} \text{ for } k_1 < k \leq K \qquad J_2 \text{ is a fresh interface}}{\langle g_1, T[x := \textbf{wait } e], k_1, d, I, J_1\rangle \rightarrow \langle g_2, T[x := v], k_2, d, I \oplus J_1, J_2\rangle}$$

$$\textsc{CDelay}\quad \frac{d < K \qquad g_1 = I[k].\mathsf{out} \qquad g_2 = I[k+1].\mathsf{out}}{\langle g_1, w, k, d, I, J\rangle \rightarrow \langle g_2, w, k+1, d+1, I, J\rangle}$$

$$\textsc{Summary}\quad \frac{v_2 \in e(g, \ell) \qquad I_1 \text{ and } J_1 \text{ are fresh interfaces} \qquad \langle I_1[k_1].\mathsf{out}, \langle v_1, s_p\rangle, k_1, 0, I_1, J_1\rangle \rightarrow \ldots \rightarrow \langle I_2[k_2].\mathsf{out}, \langle \ell, S[\textbf{return } e]\rangle, k_2, d, I_2, J_2\rangle}{\langle p, v_1, k_1\rangle \rightsquigarrow \langle I_2 \oplus J_2, d, v_2, k_2\rangle}$$

Figure 5: The compositional program semantics; though we omit them, the rules for the standard sequential statements are straightforwardly derived, according to those in Section 2 and Appendix A.

ASYNC transition of Section 2, the CASYNC rule simply combines the interface J_2 of the asynchronously-called task with the accumulated interfaces J_1 of previously-called tasks. The CWAIT rule then, by sequencing the accumulated interface J_1 of previously-called tasks before the current task's interface I, effectively fast-forwards the current task's execution to a point after the execution of the previously-called tasks, and resumes in the round k_2 in which the waited task finished. The CDELAY rule simply advances the current task to its next round, spending a single delay. Finally, the SUMMARY rule defines the interface generation relation $\rightsquigarrow$ as the composition of the task's internal interface I_2 with the accumulated interfaces J_2 of its subtasks.

We then define $\tilde{R}(P, K)$ as the set of global valuations labeling the output of completed interfaces of the main procedure:

$$\tilde{R}(P,K) = \left\{ I[k].\mathsf{out} : \begin{array}{l} \langle \mathsf{main}, \ell_0, 0\rangle \rightsquigarrow \langle I, \_, \_, k\rangle, \\ |I| = K{+}1, \text{ and } I \text{ is complete} \end{array} \right\}$$

This definition allows us to relate the global valuations reachable by executions of DFW(K) with those reached in our compositional semantics with (K+1)-round interfaces.

LEMMA 1. $R(P, \mathrm{DFW}(K)) = \tilde{R}(P, K)$.

5. SEQUENTIALIZATION

The compositional semantics of the previous section lead to an alternate way to execute asynchronous programs according the DFW(K) using nondeterministic choice (in the instantiation of fresh task interfaces): rather than adding asynchronously-called tasks to a task pool, we can simply guess the global states that a task will encounter at the beginning of each of its (up to K+1) rounds, and obtain one possible (K+1)-length interface before continuing execution of the calling task. In essence, querying a task for its interface at the point where it is called mimics the same control flow as a procedure call. In this section, we exploit this fact to generate a sequential program $\Sigma(P, K)$ which simulates a given asynchronous program P under the DFW(K) scheduler; to obtain the interface of an asynchronously-called task, $\Sigma(P, K)$ *calls* the task synchronously, with the nondeterministically-guessed global states constituting the input values of the task's interface. Figure 6 lists the statement-by-statement translation $\Sigma(P, K)$ of a program P; for simplicity, the listed translation assumes that there is one single global variable `g`; the extension to multiple global variables is straightforward, by multiplying the `G`, `Guess`, `Next`, and `Save` variables.

Our sequentialization $\Sigma(P, K)$ essentially encodes the interfaces of the previous section using the global `G`, `Guess`, and `Next` variables, along with the `Save` procedure-local variables, and the `Init` constant of the main procedure. Initially, the root task, defined by the `main` procedure, guesses the global values it will encounter at the first point at which it either returns, or waits for a task to complete; this value is stored in both `Next` and `Guess`, and corresponds to the output values of interface I in the compositional semantics of Figure 5; the input values of I are stored in `Init`. If the root procedure encounters a **`wait`** statement, then it validates its `Guess`, advances its state to `Next`, where its previously-called subtasks have left off, and guesses the next global values at which it will either return or encounter a **`wait`** statement; this process corresponds to composing the I and J_1 interfaces in the CWAIT rule of the compositional semantics, effectively sequencing the effects of previously-called tasks before resuming from the **`wait`** statement.

The other key interesting aspect of $\Sigma(P, K)$ is the translation of the **`async`** statement. Similar to the sequentialization of the DF(K) scheduler [1], the procedure of an asynchronous task is called *synchronously*, using the values `Next` of the global variables effected by previously-called asynchronous procedures; furthermore, the global values guessed to be left behind by the called task are stored into `Next`, from which subsequently-called tasks will resume.

While the global values reachable in the K-delay sequentialization $\Sigma(P, K)$ of a program P are not directly comparable to those of P, since the global variables of $\Sigma(P, K)$ are (K+1)-length vectors of values, we can compare values using a projection function φ mapping $\Sigma(P, K)$'s configurations to values of P. In particular, we define $\varphi(c)$ as `Next`$[K](c)$, i.e., the valuation of the `Next` vector's last element in c; then we define $R_\varphi(P) = \{\varphi(c) : c_0 \rightarrow \ldots \rightarrow c \text{ is finalized}\}$. Given this projection, we can show that the set of projected reachable global values in the K-delay sequentialization $\Sigma(P, K)$ of an asynchronous program P is precisely equal to the set of values reachable in P in the K-bounded compositional semantics.

```
// translation of var g: T
var G[K+1], Guess[K+1], Next[K+1]: T

// new global declarations
var delays: int

// translation of proc p(l: T) s
proc p(l: T, k: K+1)
  var Save: ([K+1]: T) * ([K+1]: T);
  s

// translation of proc main() s
proc main()
  const Init[K+1]: T := G;
  var k: int := 0;
  delays := 0;
  Next := Guess := ⋆;
  s;
  assume G = Guess;
  assume Init[1..K+1] = Next[0..K]

// translation of access to g
G[k]

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

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

// translation of async t := p e
Save := (G, Guess);
G := Next;
Next := Guess := ⋆;
call t := p(e,k);
assume G = Guess;
G, Guess := Save

// translation of x := wait t
assume G = Guess;
G := Next;
Next := Guess := ⋆;
x, k' := t; k := max(k,k')

// at each possible preemption
if (⋆ && delays < K)
  delays := delays+1; k := k+1
```

Figure 6: The K-delay sequentialization $\Sigma(P, K)$.

LEMMA 2. *$R_\varphi(\Sigma(P,K)) = \tilde{R}(P,K)$.*

Combining Lemmas 1 and 2, we have our equivalence between the valuations reachable in executions of P under the DFW(K) scheduler with those reachable in executions of the sequential program $\Sigma(P,K)$.

THEOREM 2. *$R(P, \mathrm{DFW}(K)) = R(\Sigma(P,K))$.*

6. COMPLEXITY

While Section 3 establishes that DFW(K) generally reaches more program variable valuations than DF(K) does, an obvious concern would be the cost at which it does so. In this section we demonstrate that despite the increased power of DFW(K) with respect to reachability, the essential cost of exploration is roughly equivalent, in that the reachability problem falls into the same NP-complete class as that of DF(K). As is standard in the literature, we focus on the effects on complexity arising from concurrency, factoring out effects arising from data; we thus measure the asymptotic complexity of the global-variable value reachability problem assuming program variables have finite domains, and that the number of program variables is fixed. Otherwise, general infinite data domains would lead to undecidability, and the exponential number of valuations of a non-fixed number of program variables would interfere with our complexity measurement. Formally, the DFW(K) *value reachability problem* asks whether a given global program variable valuation g of a given program P is included in $R(P, \mathrm{DFW}(K))$, for a given $K \in \mathbb{N}$, written in unary.

NP-hardness follows directly from the NP-hardness of DF(K)'s reachability problem [1], since $R(P, \mathrm{DF}(K)) = R(P, \mathrm{DFW}(K))$ for programs P without **wait** statements.

LEMMA 3. *The* DFW(K) *value reachability problem is NP-hard.*

Our proof of NP-membership reduces the problem to value reachability in sequential programs with a fixed number of variables in K. While this amounts to a sort of sequentialization, our sequentialization of Section 5 is inadequate, since $\Sigma(P,K)$ has a *linear* number of program variables in K, evaluating to an exponential number of valuations in K. The crux of our proof is thus to design a sequentialization which uses only a *constant* number of additional program variables, independently of K.

LEMMA 4. *The* DFW(K) *value reachability problem is in NP.*

Combining proofs, we have a tight complexity result.

THEOREM 3. *The* DFW(K) *value reachability problem is NP-complete.*

7. EMPIRICAL EVALUATION

We evaluate our DFW(K) scheduler empirically by comparing its sequentialization with an analogously implemented sequentialization of Emmi et al.'s DF(K) scheduler [1]; we have implemented both sequentializations in the `c2s` tool[6]. As the DF(K) scheduler does not interpret **wait** statements, we pre-process each program given to the DF(K)-based sequentialization with the translation of Figure 7, which outputs an equivalent program without **wait** statements. Essentially, this program keeps track of whether each task has finished using the global `result` variable; the translation of each **wait** statement for a task cannot proceed until its task has completed.

All of our experiments are carried out by applying a sequentialization (either DF(K)'s or DFW(K)'s) on a Boogie code representation[7] of the input asynchronous program, which is fed to the Corral verification engine [2] to detect whether an assertion violation can be reached within a given delay bound K.

[6] `https://github.com/michael-emmi/c2s`

[7] Boogie is an intermediate verification language [6].

```
// new global declarations
var result[int]: T;
var uniqueId: int

// translation of proc p(l: T) s
proc p(l: T, self: int) s

// translation of proc main() s
proc main()
result := [⊥, ⊥, ..];
uniqueId := 0;
s

// translation of call x := p e
call x := p (e,0)

// translation of return e
result[self] := e;
return e

// translation of async t := p e
t := ++uniqueId;
async p(e,t)

// translation of x := wait t
x := result[t];
assume x != ⊥
```

Figure 7: A preprocessing step for the DF(K) sequentialization to remove wait statements.

Our first set of experiments measures the delay bound and total time necessary to discover assertion violations corresponding to errors reported in a set of C# code fragments found on StackOverflow and MSDN — each around 25-50 LOC. Though we have manually translated the original C# code to Boogie, we have done so in a mechanical way which we believe, due to our experience developing mechanical translations[8] would be roughly equivalent to an automatic translation; the C# and Boogie sources of the examples, along with a script to replicate our experiments, is on Github[9].

Figure 8 shows Corral's execution time to reach each assertion violation in the DF(K) and DFW(K) sequentializations. In each run, we begin with the delay bound $K = 0$ and increase K until the assertion violation is reachable in the sequentialized program. Our results demonstrate that the DFW(K) scheduler requires consistently fewer delays to reach the assertion violations, which amounts to less exploration time in Corral. The biggest differences appear in the first and third examples, in which the assertion violation is preceded by chains of sequenced asynchronous calls — i.e., where each asynchronous call in the chain is only made after the previous one is waited for; intuitively, each link in this chain forces DF(K) to spend another delay just to progress its execution, whereas DFW(K)'s natural scheduling order proceeds past each link without spending a delay. These examples illustrate that such call chains are commonplace; even the small bit of code in the third example contains a chain of 5 calls.

In order to validate the efficacy of our delay-bounded sequentialization approach, we have also implemented a "depth-bounded" exploration by translating (by hand) the first `CollectionLoad` example into a sequential program which

[8] `https://github.com/smackers/smack`.
[9] `https://github.com/michael-emmi/sync-aware-experiments`

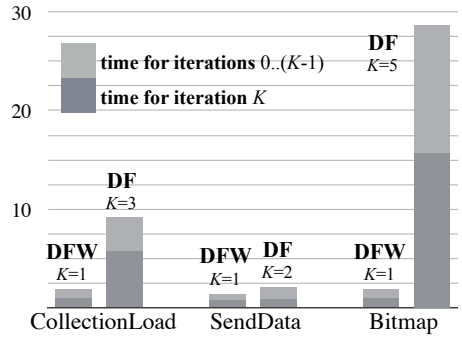

Figure 8: Time to bug detection (in seconds for three examples using the DFW and DF sequentializations. Each bar represents the aggregate time over increasing delay bounds, starting from zero, whereas the dark part indicates time spent for the smallest successful delay bound (K).

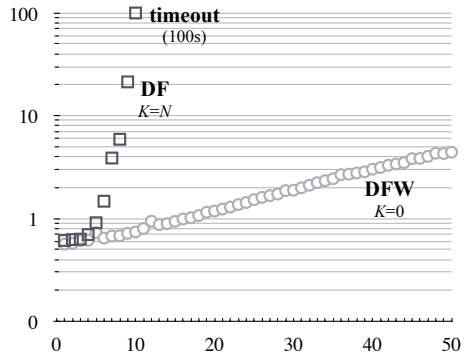

Figure 9: Time to bug detection (in seconds) for the parameterized example with N (on the X-axis) chained asynchronous calls. While the DFW sequentialization consistently discovers the bug without delays, DF requires $K = N$ delays, and times out at 100s for $N = 10$.

simulates every asynchronous program execution up to a given number of program steps — we consider that each program statement constitutes one program step. This program's top-level procedure contains a loop in which each iteration executes a single step of a nondeterministically-chosen task; K iterations of this top-level loop thus simulates all possible asynchronous program executions with up to K steps. Exploration of this program with Corral is intractable: the same bug discovered with DFW(1) requires $K = 9$ program steps, yet Corral is only able to explore up to $K = 4$, in 90 seconds, before timing out at 100 seconds for any depth $K \geq 5$. Note that while DFW(K) is practically limited by the degree K of deviation from DFW(0), of which small values seem to suffice in exposing concurrency errors, DFW(K) is not inherently limited by execution depth.

Our second set of experiments attempts to measure the effect of the aforementioned asynchronous call chains on the DF(K) and DFW(K) sequentializations using a very simple parameterized program $P(N)$: for each $N \in \mathbb{N}$, $P(N)$ makes N asynchronous calls (to a procedure which simply returns)

waiting for each before calling the next, ultimately followed by an assertion violation — i.e., **`assert false`**. As Figure 9 illustrates, the DFW(K) scheduler never requires a delay to reach the assertion, and its sequentialization scales well, with Corral completing in under 5 seconds even for chains of 50 calls. The DF(K) scheduler, however, requires N delays for each chain of N calls, and times out at 100 seconds without completing for chains of 10 calls. The utter simplicity of the program $P(N)$ suggests that the DF(K) sequentialization is limited to very small chains, and ultimately small fragments of synchronization-heavy programs.

8. RELATED WORK

Our work follows the line of research on compositional reductions from concurrent to sequential programs. The initial so-called "sequentialization" [7] 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 [4, 5]. La Torre et al. [8] later extended the approach to handle programs parameterized by an unbounded number of statically-determined threads, and shortly after, Emmi et al. [1] 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]. Bouajjani et al. [3] pushed these results even further to a sequentialization which attempts to explore as many behaviors as possible within a given analysis budget. While others have continued to propose sequentializations for other bounded concurrent exploration criteria or program models [10, 11, 12, 13, 14, 15], as far as we are aware, none of these sequentializations is based on a parameterized scheduler which can reduce exploration cost by taking into account program synchronization.

9. REFERENCES

[1] Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL '11: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM (2011) 411–422

[2] Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV '12. Volume 7358 of LNCS. 427–443

[3] Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: SAS '11: Proc. 18th International Symposium on Static Analysis. Volume 6887 of LNCS., Springer (2011) 129–145

[4] Qadeer, S., Rehof, J.: 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., Springer (2005) 93–107

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

[6] Barnett, M., Leino, K.R.M., Moskal, M., Schulte, W.: Boogie: An intermediate verification language `http://research.microsoft.com/en-us/projects/boogie/`.

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

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

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

[10] Kidd, N., Jagannathan, S., Vitek, J.: 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., Springer (2010) 245–261

[11] Garg, P., Madhusudan, P.: Compositionality entails sequentializability. In: TACAS '11: Proc. 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 6605 of LNCS., Springer (2011) 26–40

[12] Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. In: TACAS '12: Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, Springer (2012)

[13] Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: CAV '12: Proc. 24th International Conference on Computer Aided Verification. LNCS, Springer (2012)

[14] Emmi, M., Lal, A.: Finding non-terminating executions in distributed asynchronous programs. In: SAS '12: Proc. 19th International Static Analysis Symposium. LNCS, Springer (2012)

[15] Emmi, M., Lal, A., Qadeer, S.: Asynchronous programs with prioritized task-buffers. In: SIGSOFT FSE '12: Proc. 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM (2012) 48

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

[17] La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing parameterized programs (2010) Under submission.

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

[19] Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: POPL '12: Proc. 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM (2012) 203–214

[20] Sharir, M., Pnueli, A.: Two approaches to interprocedural data-flow analysis. In Muchnick, S.S., Jones, N.D., eds.: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981) 189–234

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

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

[23] Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. **34**(1) (2012) 6

APPENDIX

A. PROGRAM TRANSITIONS FOR SEQUENTIAL STATEMENTS

Transition rules for the standard sequential program statements in an asynchronous program are listed in Figure 10.

The SKIP rule just proceeds to the next statement. The ASSUME rule proceeds only if the expression e evaluates to true. The THEN rule proceed to the then branch if the current valuation of the given expression e evaluates to true, ELSE rule proceeds to the else branch if e evaluates to false. The LOOP rule iterates the loop if the given expression e evaluates to true, and BREAK skips the loop if e evaluates to false. The GLOBAL/LOCAL statements set the value of a global/local variable to the value that given expression evaluates to.

$$\textsc{Skip}\ \frac{}{C[\mathbf{skip}; s] \to C[s]} \qquad \textsc{Assume}\ \frac{\mathbf{true} \in e(C)}{C[\mathbf{assume}\ e] \to C[\mathbf{skip}]}$$

$$\textsc{Then}\ \frac{\mathbf{true} \in e(C)}{C[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to C[s_1]}$$

$$\textsc{Else}\ \frac{\mathbf{false} \in e(C)}{C[\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2] \to C[s_2]}$$

$$\textsc{Loop}\ \frac{\mathbf{true} \in e(C)}{C[\mathbf{while}\ e\ \mathbf{do}\ s] \to C[s; \mathbf{while}\ e\ \mathbf{do}\ s]}$$

$$\textsc{Break}\ \frac{\mathbf{false} \in e(C)}{C[\mathbf{while}\ e\ \mathbf{do}\ s] \to C[\mathbf{skip}]}$$

$$\textsc{Global}\ \frac{x \text{ is a global variable} \qquad v \in e(g, M)}{\langle g, M[x := e]\rangle \to \langle g(x \mapsto v), M[\mathbf{skip}]\rangle}$$

$$\textsc{Local}\ \frac{\begin{array}{c}x \text{ is a local variable} \\ v \in e(g, \ell) \qquad \ell_2 = \ell_1(x \mapsto v)\end{array}}{\langle g, m \cup \{\!\{i, \langle \ell_1, S[x := e]\rangle \cdot w, v\}\!\}\rangle \to \langle g, m \cup \{\!\{i, \langle \ell_2, S[\mathbf{skip}]\rangle \cdot w, v\}\!\}\rangle}$$

Figure 10: Transition rules for sequential statements.

B. PROOFS TO SELECTED RESULTS

$R(P, \mathrm{DFW}(K)) = \tilde{R}(P, K)$

The asynchronous semantics under DFW(K) and the compositional semantic are equivalent. Figure 11 shows the construction of the interface I that summarizes exactly execution produced by DFW(1) scheduler (given in in Figure 11(a)). Consider Task A, that consequently creates tasks B and C, waits for B (at the end of segment 1) and waits for C (at the end of segment 4). C is a delaying task (delays after segment 3) and completes after segment 5.

Basically, A updates the current state it executes in (following the standard sequential semantics). Each time it creates a task, it collects and sequences together the interfaces of these created tasks into J (see the sequencing of the interfaces of B and C in Figure 11(b)). When it waits for a task, it sequences the accumulated tasks in J into I (as in Figure 11(c). Then, it continues with its internal execution. When A waits for the delaying task C, J does not keep new posted tasks to be joined into I, hence I is not updated. But A continues sequencing the rest of its execution (segment 6) in a later round in which C completes. When A completes, it ends up with the interface I that summarizes the execution in Figure 11(a).

We can show the equivalence between the DFW(K) execution and the compositional semantics inductively. Consider building the execution of a program P. Initially, P starts in round 0 with an initial state g_0, having $I[0].\mathsf{in} = g_0$. At each statement of P, the compositional semantics builds the interface I of P iteratively.

- At each execution of a sequential statement, it updates its current state $g \to g'$ yielding $\langle g, w, k, d, I, J\rangle \to \langle g', w, k, d, I, J\rangle$.
- At each task creation, it creates a new task interface summarizing its child task and joins this new child task's interface to the collected tasks. The join operation ensures that this new task starts its execution where the last posted task has left off. Hence, it simulates the execution of this task in the ordering that DFW(K) provides. As a result of the join operation, out of J now keeps the effects of this last posted task. Sequencing an interface of another task to J will simulate that task to start from where the last task in J ends in.
- When a task waits, the cumulated tasks in J update I by appending to it, enabling the next statements of the program to operate on the state reached by executing the tasks in J. The transition also moves the current task to the round and the state where the waited task has completed. This simulates the execution under DFW(K) that schedules the waiting task in the round that the waited task completes, after the execution of all its children tasks. Notice that the input and the output intermediate states of I are equal, since the task waits and does not involve in the execution until the waited task completes.
- When a task is delayed, CDELAY assigns the current state to the out state of the interface, and moving the next state in a later round.

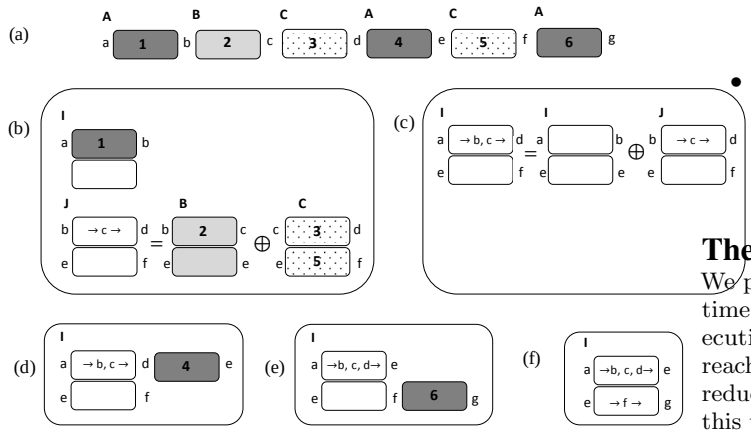

Figure 11: Simulating an asynchronous program using compositional semantics.

At the end of the execution, $I[k]$.out keeps the state reached in at most k delays.

$R_\varphi(\Sigma(P,K)) = \tilde{R}(P,K)$
Here, we prove that for an asynchronous program P, our translation precisely produces the compositional semantics and the set of reachable global values in the K-delay sequentialization $\Sigma(P,K)$ of an asynchronous program P is precisely equal to the set of values reachable in P in the K bounded compositional semantics.

Our sequentialization actually encodes the interfaces in the compositional semantics using `G`, `Guess`, `Next` and `Save` variables. Essentially, `Next` keeps J.out and gets updated each time J is joined with a newly posted task. To be able to append the posted tasks after the internal task that creates them, initially `Next` is set to the state it pauses (in either a **wait** or **return** statement) or delays. Since do not know that state initially, we guess the ending states of each round and keep in `Guess`. When we reach to that pause/delay point, we validate the guess.

- The CAsync rule is encoded by the translation of the **async** statement. The translation sets the current state `g` to the `next` state where the last posted task has finished with (corresponding to the J.out). It updates the `next` variable with the new task's ending state which is guessed before its execution and it is verified to be the current state after the execution of the posted task. This precisely captures the generation of a new interface for the new task and joining it with the accumulated task interface J.
- CWait rule is encoded by the translation of **wait** statement. The translation verifies the guess that keeps the pause state of the current task to be the current state and sets the state to `next` that keeps the ending state of the last posted task. This has the same behavior with the CWait rule as it joins the current interface with J (ensuring that the out state of the I is equal to the in state of J). Since the translation does not touch the states `g[i]` in the intermediate rounds, it satisfies the equality of the in and in states of these intermediate rounds.
- CDelay rule is encoded in the translation of preemption that increases the current round. Since the program reads the value of the `g` as `g[i]` in a given round i, it already operates on the state corresponding to its current round.

The DFW(K) value reachability problem is in NP.

We prove membership in NP by constructing a polynomial time algorithm that verifies a guessed polynomial-sized execution witness in polynomial time. The verifier program reaches its final state if the provided witness is valid, hence reduces our NP-membership problem to the reachability in this verifier program.

A witness represents a K-delaying execution is a list of delaying task interfaces in an asynchronous program P given in the depth-first traversal order of the task creation tree. Here, we extend a task interface to include the procedure that the task executes together with its parameter, the round it starts its execution in, total number of tasks it posts and the indices of this children in the witness in order.

The construction of the verifier program uses a similar symbolic encoding with our sequentialization and given in Figure 12.

The program uses the witness delaying tasks and the total number of delays as the global variables and calls the verifier (the translated procedure) of each task separately. Each procedure checks whether: (i) the execution of this task yields the g.in and g.out states of its interface and (ii) it posts all its children tasks in the given order.

Similar to the encoding of sequentialization, translated procedure keeps `save`, `guess` and `next` states in addition to the current round it executes in and the number of tasks it has posted. If non-delaying task is created, we just update the next state applying its effect. If it is a delaying task, we first check whether the posted procedure is the procedure of the next task to be executed and current round is the start round of this task. Also, the interface of this task is checked whether g.in state match g.out state of the last executed task (if this is the first posted task, it should match the end state of the creator task).

When a task delays, the current round is incremented and the state is set to the initial state of that round. Moreover, the next state is updated by applying the effects of *numPosted* tasks (a new task will begin after the execution of the creator task and the *numPosted* number of tasks posted in earlier rounds).

When a task returns, we verify whether it has posted all its children tasks. The out state of that round is calculated by appending the children tasks' effects to the current state and checked whether it is equal to the g.out of this task. If the task returns in a round earlier then k, the rest of the task interface is filled in with the children tasks' interfaces and g.in and g.out states of children are validated whether they correspond to each other. The interface of the execution is valid if these interfaces match and the task has created the

numPosted number of tasks.

C. A STUDY OF ASYNCHRONOUS PROGRAMMING ERRORS

In this section we study classes of common asynchronous programming errors, as found in blog posts, online discussion forums such as StackOverflow, and MSDN's online documentation. The presented examples are written in C#5.0, having two new asynchronous programming constructs **async** and `await`. An asynchronous procedure in C# is a non-blocking procedure that begins its execution synchronously and may continue asynchronously when it meets an await statement. If an `await` statement waits for a task which is not completed, it registers the rest of the execution as a callback of the waited task and returns control of execution to its caller. The semantics of the asynchronous procedure in C# (whose part of statements execute concurrently to its caller) is different then a task creation that posts a task to be executed completely in parallel to its caller. Though, our method can still be used to detect possible bugs in these programs (with a possibility of false positives) as we provide in 7. (Note that C# also provides constructs for task creation and blocking wait.)

C.1 Bugs due to Unmanaged Side Effects

A `await` statement makes sure that the rest of the code after this statement does not execute until the waited task completes. However, it does not provide any guarantees to the caller of this method. While writing asynchronous programs, programmer may neglect the fact that an asynchronous task (executed by a non-blocking call) may not be completed when the control of execution returns to the caller. The programmer should be aware of the side effects of non-blocking calls methods when some data are expected to be returned or set. A convenient way to prevent these bugs is to make sure that the task whose effect is required has finished (i.e. checking whether it has completed or waiting for it) before the rest of the statements. To be able to **wait** or `await` for a task, that non-blocking method should return a Task handle instead of void. There are many discussions related to this category of bugs and real buggy code samples on the web blogs and programming forums.

Example 1. A Windows store app example on MSDN

```
string m_GetResponse;
async void Button1_Click(object Sender, EventArgs e)
{
  try{
    SendData("https://secure.flickr.com/services/oauth/request_token");
    await Task.Delay(2000);
    DebugPrint("Received␣Data:␣" + m_GetResponse);
  }
  catch (Exception ex) {
    rootPage.NotifyUser("Error␣posting␣data␣to␣server." + ex.Message);
  }
}

async void SendData(string url)
{
  var request = WebRequest.Create(url);
  using (var response = await request.GetResponseAsync())
    using (var stream = new StreamReader(response.GetResponseStream()))
    m_GetResponse = stream.ReadToEnd();
}
```

In the `Button1_Click` method, non-blocking `SendData` method is called and it is waited for two seconds (since the printing statement executes after the delaying task completes after two seconds). The printing statement uses a variable that will be set in `SendData`. `SendData` method, creates a request and `await`s for a non-blocking method that will return a response. Notice that, if the response is not ready, `SendData` method returns to its caller when it `await`s GetResponseAsync. After getting the response, the rest of the body executes as a callback and sets `m_GetResponse`. In case it takes longer than two seconds to complete `SendData`, the contents of `m_GetResponse` will not be ready.

The code can be corrected by awaiting for `SendData` method before using `m_GetResponse` in its caller method. Since `SendData` is "**async void**" and does not return a task, it cannot be awaited. So, `SendData` should be modified so that it returns a task and is awaited instead of awaiting Task.Delay. Another problem of the code is since it does not return a task, it is not possible to catch the exceptions.

Example 2. from a post on MSDN Forums

```
public App()
{
  this.InitializeComponent();
  this.VM = new ViewModel();
}

protected async override void OnLaunched(LaunchActivatedEventArgs args)
{
  Frame appFrame = new Frame();
  SuspensionManager.RegisterFrame(appFrame, "AppFrame");
  appFrame.Navigate(typeof(CategoriesPage), this.VM);

  Window.Current.Content = appFrame;
  Window.Current.Activate();
}

public ViewModel()
{
  this.InitializeData();
}

private async void InitializeData()
{
  this.InvFileName = "NVIN_Home.xml";
  await this.DeserializeDataSetAsync();
}

private async Task DeserializeDataSetAsync()
{
  StorageFolder storageFolder = ApplicationData.Current.LocalFolder;
  var dcs = new DataContractSerializer(typeof(ObservableCollection<Cate
  using (var stream1 = await storageFolder.OpenStreamForReadAsync(this.
    this.Categories = (ObservableCollection<Category>)dcs.ReadObject(st
}
//Categoriespage code
//When we arrive here, this.VM.Categories may be null
protected override async void LoadState(Object navigationParameter,
  Dictionary<String, Object> pageState)
{
  this.VM = navigationParameter as ViewModel;
  this.itemsViewSource.Source = this.VM.Categories;
}
```

The programmer calls `ViewModel` constructor in the `App` method. `ViewModel` calls a non-blocking method `InitializeData` which awaits `DeserializeDataAsync`. Upon waiting, this method returns the control of execution to its caller. In some

executions of the program, it is probable that the execution of `DeserializeDataAsync` (which also calls another nonblocking method `OpenStreamForReadAsync` has not finished when its caller resumes execution. In such a case, the categories collection is not loaded when it is used.

C.2 Bugs due to Unexpected Asynchronous Activity

In the first category of bugs, the programmer knows that there are some concurrent parts in his program. In some cases (e.g. when the programmer uses some library methods) he may not be aware of the asynchronous activity that his program exhibits. An application programmer can know whether a library method is blocking or non-blocking by looking at the method signature. However, a method not declared as **`async`** may involve some asynchronous activity, i.e. return to its caller asynchronously without completing its job. Consider a library method that calls an **`async void`** non-blocking method inside. The caller does not await for this method since it does not return a task. Since the method does not contain an **`await`** statement, it will not be declared as **`async`** (C# requires methods having **`await`** (in C#, **`await`**) statement to be declared as **`async`**). From the programmer's point of view, this library method is synchronous (i.e. blocking) and he can use it by assuming that all of the work in this method has completed when the control returns to his program. Note that, a method having some asynchronous activity can be declared as synchronous when (i) it does not **`await`** for the asynchronous method (possibly an **`async void`** method that does not cause a warning when does not await) or (ii) it has a blocking **`wait`** method that blocks the execution of this method until the waited task is finished. While the first case may result in bugs caused by the hidden concurrency, the latter case may cause deadlocks depending on the thread it blocks.

Example 3. from a post on StackOverflow

```
// MySubClass
BitmapImage m_bmp;
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
  base.OnNavigatedTo(e);
  await PlayIntroSoundAsync();
  image1.Source = m_bmp;
  Canvas.SetLeft(image1, Window.Current.Bounds.Width - m_bmp.PixelWidth);
}
protected override async void LoadState(Object nav, Dictionary<String, Object> pageState)
{
  m_bmp = new BitmapImage();
  var file = await StorageFile.GetFileFromApplicationUriAsync("ms-appx:///pic.png");
  using (var stream = await file.OpenReadAsync())
    await m_bmp.SetSourceAsync(stream);
}
// base class
class LayoutAwarePage : Page
{
  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
              // ...
    this.LoadState(e.Parameter, null);
  }
}
```

`OnNavigatedTo` and `LoadState` are overriden methods and made non-blocking. In `OnNavigatedTo`, the programmer calls the base method and `await`s for a non-blocking `PlayIntroSound` method. Then, he uses the image which is set in `OnNavigatedTo` method. The problem here is that, the base `OnNavigatedTo` method has a non-blocking call to the overriden `LoadState` method which may return before completing its job. In turn, the base `OnNavigatedTo` also returns before completion. In such a case, the image will not be set and length and width of the image will be zero.

C.3 Bugs due to Task-Buffer Nondeterminism

Each programming language gives different semantics to task buffer in which the posted tasks are collected. Consider the following cases for a single serial and/or FIFO task buffer:

- Tasks in the buffer execute serially (one after the other) in FIFO order: The execution is equivalent to a sequential execution of the tasks in the order they are posted.
- Tasks in the buffer execute serially but not guaranteed to be taken in FIFO order: In this case, tasks can be considered to execute atomically, but their dispatching order is nondeterministic. The execution may be problematic if the tasks involve non-commutative work, since the result of the whole execution depends on the dispatching order of tasks.
- Tasks on a FIFO task buffer do not execute serially: Although the tasks are dispatched in the order they are posted, their execution is concurrent since the tasks can interleave. The execution of tasks can interfere with shared memory accesses and bugs can arise from data races.

C# thread pool maintains a global task queue (for top level tasks) in FIFO order and a local task queue (for nested tasks that are created in the context of other tasks) in LIFO order. When a thread is available, it takes a task from a task queue (first checks the local queue and then the global one) and executes it. By implication, the posted tasks can be executed in parallel by different threads. The execution of C# tasks can be considered in the third case, having an ordered buffer and allowing concurrent execution of tasks.

C.4 Bugs due to Complicated Control Flow

Control flow bugs are caused when undesired flow of executions are allowed by the program's control flow. It is harder to follow and reason about in asynchronous programs that create tasks, do non-blocking method calls, wait for their result and post callbacks.

The following example, the programmer refactors his code by dividing a method into two methods. Although it seems to be a minor modification, the control flow of the new program changes and allows for some buggy executions. This example demonstrates the fact that it is harder to follow the control flow in asynchronous programs. If a programmer is not aware of the new way of thinking and analysis for asynchronous programs, he can unintentionally introduce bugs into his programs.

Example 4. from the blog of Nordic Software Company

```
async void AcquireFromCamera1(object sender, RoutedEventArgs e)
{
  try{
    var imageStream = await _cameraCapture.Shoot();
    var dto = new Dto(){ImageStream = imageStream};
    dto.Id = Guid.NewGuid().ToString();
    var file = await _fileHndlr.CreateFileAsync(dto.Id);
    dto.ImageFilePath = file.Path;
    _fileOperator.StoreStream(dto.ImageStream, file);
    SaveNewDataItem(dto);
    var dataItem = dataSource.GetItem(dto.Id);
    StoreData(dataItem);
    Frame.Navigate(typeof(EditDataPage), dto.Id);
  } catch (Exception ex)
  {
    new MessageDialog(ex.Message).ShowAsync();
  }
}

async void AcquireFromCamera2(object sender, RoutedEventArgs e)
{
  try{
    var imageStream = await _cameraCapture.Shoot();
    var dto = new Dto(){ImageStream = imageStream};
    _handler.ImageCaptured(dto);
    Frame.Navigate(typeof(EditDataPage), dto.Id);
  }
  catch (Exception ex)
  {
    new MessageDialog(ex.Message).ShowAsync();
  }
}

async void ImageCaptured(Dto dto)
{
  dto.Id = Guid.NewGuid().ToString();
  var file = await _fileHndlr.CreateFileAsync(dto.Id);
  dto.ImageFilePath = file.Path;
  _fileOperator.StoreStream(dto.ImageStream, file);
  SaveNewDataItem(dto);
  var dataItem = dataSource.GetItem(dto.Id);
  StoreData(dataItem);
}
```

AcquireFromCamera and ImageCaptured methods separate the concerns of storing and saving files and meta data for the image performed together in the original method. But this refactoring introduces a bug. In this version, while the asynchronous procedure CreateFileAsync is executing, the control of execution returns to its caller AcquireFromCamera. Hence, it is possible that Frame.Navigate executes before the completion of ImageCaptured. The caller of the non-blocking method is changed in a way that it allows for some undesired executions.

```
Task{
  startRound: int
  proc: Proc x E
  gin, gout: G^k
  numChildren: int
  children: int^numChildren
}

// Globals:
var tasks: Task^k;
var k: int;
Top(witness: Task^k)
  tasks := witness;
  for i=0 to k-1
    call (tasks[i].proc)';

// proc (e)
proc (e, curr)
  var save, guess, next: G
  var numPosted, R: int;
  numPosted := 0;
  R := tasks[curr].startRound;
  s';

// post p e
if * && <p,e> =
tasks[curr].children[numPosted].proc &&
R = tasks[numPosted].startRound then
  assume next = tasks[numPosted].gin[R];
  next := tasks[numPosted].gout[R];
  numPosted ++;
else
  save := g;  g := next;
  guess := next := *;


  call p e;
  assume guess = g;  g := save;
// delay:
assume guess = g;
next := guess := *;
R++;

if numPosted > 0 then
  for i=0 to numPosted -1
    assume next = tasks[children[i]].gin[R];
    next := tasks[children[i]].gout[R];
    g = current gin[R];

// return
assume guess = g;
assume numPosted = numChildren;

if numChildren > 0 then
  for i=0 to numChildren-1
    assume next = tasks[children[i]].gin[R];
    next := tasks[children[i]].gout[R];
assume tasks[curr].gout[R] = next;

while R< k-1
  R++;
  assume tasks[curr].gin[R] =
    tasks[children[0]].gin[R];
  for i=0 to numChildren-1
    assume tasks[children[i]].gout[R] =
      tasks[children[i+1]].gin[R];

assume tasks[numChildren-1].gout[R]
  = current.gout[R];
```

Figure 12: Translation to construct the verifier program