Exploiting Synchronization in the Analysis of Shared-Memory Asynchronous Programs*

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

Burcu Kulahcioglu Ozkan
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 limited sets of asynchronous task schedules that are likely to expose programming errors. These approaches are based on *parameterized* task schedulers, each of which admits schedules which are variations of a default deterministic schedule. By increasing the parameter value, a larger variety of schedules is explored, at a higher cost. The efficacy of these approaches depends largely on the default deterministic scheduler on which varying schedules are fashioned.

We find that the limited exploration of asynchronous program behaviors can be made more efficient by designing parameterized schedulers which better match the inherent 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, which leverages existing (sequential) program analysis tools by encoding asynchronous program executions, according to a particular scheduler, as the 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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

*This work was partially funded by Microsoft Research Outreach and the Scientific and Technological Research Council of Turkey (TUBITAK).

SPIN '14, July 21–23, 2014, San Jose, CA, USA

General Terms

Algorithms, Reliability, Testing, Verification

Keywords

Concurrency, Asynchronous programs, Sequentialization

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 (`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)

```
// 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.

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 $\mathrm{DF}(K)$ to further deviate from its natural deterministic order by increasing K. For instance, if $\mathrm{DF}(K)$'s default schedule encounters a statement which acquires a lock held by another thread, then $\mathrm{DF}(K)$ 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, $\mathrm{DF}(K)$ 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 $\mathrm{DF}(K)$ only for large values of K. As the cost of program exploration with $\mathrm{DF}(K)$ 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 $\mathrm{DFW}(K)$ for which the cost of exploration is not tied to program synchronization, and yet which still enjoys $\mathrm{DF}(K)$'s strengths, in particular:

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

However, unlike $\mathrm{DF}(K)$, the $\mathrm{DFW}(K)$ 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 $\mathrm{DFW}(K)$ provides strictly greater behavioral coverage than $\mathrm{DF}(K)$ at virtually no additional cost.

Our contributions and outline are:

§2 A formal semantics of asynchronous programs with synchronization.

§3 A formal description of the $\mathrm{DFW}(K)$ scheduler, and comparison with $\mathrm{DF}(K)$.

§4 A "compositional semantics" for $\mathrm{DFW}(K)$, which fosters sequentialization.

§5 A code translation encoding $\mathrm{DFW}(K)$ executions as a sequential program.

§6 NP-completeness of reachability under $\mathrm{DFW}(K)$ for finite data programs.

§7 An empirical comparison: $\mathrm{DFW}(K)$ finds bugs faster, with smaller K.

In theory, every program behavior observable with $\mathrm{DF}(K)$ is also observable with $\mathrm{DFW}(K)$ for any K, yet the reverse is untrue: for any K_0 there exist programs whose sets of behaviors observable with $\mathrm{DFW}(K_0)$ are not all observable with $\mathrm{DF}(K)$ for any K; Section 3 demonstrates this fact. Empirically, Section 7 demonstrates that our sequentialization of $\mathrm{DFW}(K)$ is more effective than $\mathrm{DF}(K)$ 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

[1]This complexity assumes program variables are fixed in number and size, as usual [2, 1].

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 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-exploiting 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.

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

$$\begin{array}{l} s ::= s\texttt{;}\ s \mid \textbf{assume}\ e \mid \textbf{skip} \\ \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \textbf{do}\ s \\ \mid x\ \texttt{:=}\ e \mid \textbf{call}\ x\ \texttt{:=}\ p\ e \mid \textbf{return}\ e \\ \mid \textbf{async}\ x\ \texttt{:=}\ p\ e \mid x\ \texttt{:=}\ \textbf{wait}\ e \end{array}$$

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* if $i = \perp$. 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 $S ::= \diamond \mid S; s$, where $s \in \mathsf{Stmts}$. We write $S[s]$ for the statement obtained by substituting a statement s for the unique occurrence of $\diamond$ in S. Intuitively, a context substituted by s, e.g., $S[s]$, indicates that s is the next statement to execute in the statement sequence $S[s]$. Similarly, a *task context* $T = \langle \ell, S \rangle \cdot w$ 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 $\langle \ell, S[s] \rangle \cdot w$. Finally, we write $e(g, \ell)$ (resp., $e(g, T)$) to denote the evaluation of a program expression e given the global and local variable valuations $g, \ell : \mathsf{Vars} \to \mathsf{Vals}$ (resp., where ℓ is the topmost local variable valuation of T); $e(g, \ell) \subseteq \mathsf{Vals}$ is a *set* of values since program expressions may be nondeterministic, using $\star$.

Figure 2 defines an operational program semantics via a set of transition rules on program configurations; the remaining transition rules for sequential program statements are standard. 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-created 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 has 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

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

[3]We denote the empty sequence with ε.

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

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

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

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

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

Figure 2: Transition rules over program configurations.

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 an `exit` flag to simulate the control flow of an uncaught program exception [3].

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-exploiting scheduler $\mathrm{DFW}(K)$.

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 $\pi(q) \in \mathsf{IDs}$ that is enabled to make a program transition. A transition $\delta(q, i, c_1, c_2) = q'$ determines the scheduler's successor state q' to a program transition $c_1 \to c_2$ of enabled task i from scheduler state q. We represent nondeterminism using ε-transitions $\delta(q, \varepsilon)$; these transitions affect scheduler state only, and not program configuration otherwise. We say Ψ is *deterministic* when $\delta(q, \varepsilon) = q$ for all $q \in Q$.

As an example, we could define a completely nondeterministic scheduler $\langle Q, q_0, \delta, \pi\rangle$, which always enables all pending tasks, with states $Q = \mathsf{IDs}^*$ represent scheduling queues, having initial state $q_0 = \bot$; the task-selection predicate $\pi(i \cdot \_) = i$ selects the task at the head of the queue. Transitions modify the queue accordingly: enqueueing created tasks j on Async transitions $c_1 \to c_2$, $\delta(i \cdot I, i, c_1, c_2) = i \cdot I \cdot j$; otherwise not modifying the queue, $\delta(i \cdot I, \_, \_, \_) = i \cdot I$; and rotating the queue on ε-transitions, $\delta(i \cdot j \cdot I, \varepsilon) = j \cdot I \cdot i$. By making a sequence of ε-transitions, this scheduler can enable any previously-created task.

To define the executions admitted by a scheduler Ψ, we make Ψ follow program transitions, and allow Ψ to make ε-transitions at any time. Formally, 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 $f : \mathbb{N}^{<j} \to \mathbb{N}^{<j'}$ such that for each transition $c_i \to c_{i+1}$ of task u_i, for $0 \le i < j$, u_i is enabled by Ψ: $u_i = \pi(q_{f(i)})$; and the state of Ψ follows the transition $c_i \to c_{i+1}$: $\delta(q_{f(i)}, u_i, c_i, c_{i+1}) = q_{f(i)+1}$; additionally, intermediate Ψ-states follow ε-transitions: $q_{i+1} = \delta(q_i, \varepsilon)$ for $0 \le i < j'$ where $i \notin \mathrm{range}(f)$. Finally 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 has called asynchronously, 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 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 $\langle \{\bot\}, \emptyset, \{\bot \mapsto \langle \mathsf{R}, 0\rangle\}, 0\rangle$.
- $\pi(q)$ is the least, in depth-first order, minimal-round ready vertex as in Figure 3(a), and is undefined when q does not contain such a vertex.
  - $\delta(q, i, \_, \_)$ is obtained from q for Async transitions creating task j by adding a rightmost child $(j \mapsto \langle \mathsf{R}, k\rangle)$ to the $\langle \mathsf{R}, k\rangle$-labeled vertex i, as in Figure 3(b).
  - $\delta(q, i, \_, \_)$ is obtained from q for Complete transitions by updating i's label from $\langle \mathsf{R}, k\rangle$ to $\langle \mathsf{C}, k\rangle$.
  - $\delta(q, i, \_, \_) = q$ otherwise.
  - $\delta(q, \varepsilon)$ is obtained from q by incrementing the delay counter d, and updating $\pi(q)$'s label from $\langle \mathsf{R}, k\rangle$ to $\langle \mathsf{R}, k+1\rangle$, so long as $d < K$; otherwise $\delta(q, \varepsilon) = q$.

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-round tasks in depth-first order according to the task

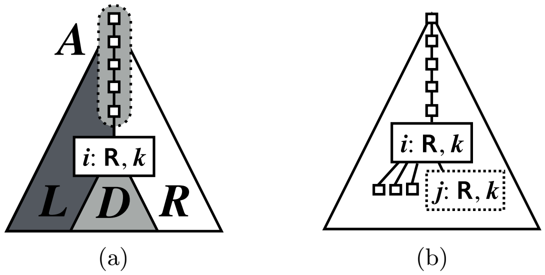

Figure 3: (a) A tree of the $\mathrm{DF}(K)$ scheduler enabling task i, showing i's ancestors (A), descendants (D), and the left (L) and right (R) descendants of i's ancestors. As i is enabled, each node in $A \cup L$ is either completed or has round $> k$, and each node in $D \cup R$ is either completed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

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 $\mathrm{DF}(K)$ 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 $\mathrm{DF}(K)$ 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 that does not enable tasks that are blocked waiting for others to complete. We define the *Synchronization-Exploiting Depth-First Scheduler* $\mathrm{DFW}(K) = \langle Q, q_0, \delta', \pi \rangle$ by extending $\mathrm{DF}(K)$ to label each tree node i with an additional Waiting status label, and a counter for the number of tasks that i has posted since its last-encountered wait statement. (While it is possible to define a "synchronization-exploiting" scheduler without this counter, the resulting sequentialization would be more complicated.) As in $\mathrm{DF}(K)$, the task-selection predicate $\pi(q)$ selects the least, in depth-first order, minimal-round ready vertex of q. We define the transition function $\delta'(q, i, c_1, c_2) \stackrel{\text{def}}{=} f(\delta(q, i, c_1, c_2), c_2)$ by the composition of transition function δ with post-processing function $f : (Q \times \mathsf{Configs}) \to Q$ as follows:

- $\delta(q, i, \_, \_)$ is obtained from q for ASYNC transitions creating task j by adding a rightmost child $(j \mapsto \langle R, k, 0 \rangle)$ to i, as in Figure 3(b), and updating i's label from $\langle R, k, a \rangle$ to $\langle R, k, a+1 \rangle$.
- $\delta(q, i, \_, \_)$ is obtained from q for WAIT transitions by updating i's label from $\langle R, k, a \rangle$ to $\langle W, k, 0 \rangle$.
- $\delta(q, i, \_, \_)$ is obtained from q for COMPLETE transitions by updating i's label from $\langle R, k, a \rangle$ to $\langle C, k, a \rangle$.
- $\delta(q, i, \_, \_) = q$ otherwise.
- $\delta(q, \varepsilon)$ is obtained from q by incrementing the delay counter d, and updating $\pi(q)$'s label from $\langle R, k, a \rangle$ to $\langle W, k+1, a \rangle$, so long as $d < K$; otherwise $\delta(q, \varepsilon) = q$.
- $f(q, c)$ updates the label of each $\langle W, k, a \rangle$-labeled vertex i to $\langle R, \max(k, k'), a \rangle$ if and only if (i) i is waiting[4] for a $\langle C, k', \_ \rangle$-labeled task, or i is not waiting and $k = k'$, and (ii) the only $\langle \_, \max(k, k'), \_ \rangle$-labeled descendants of i are descendants of i's a-rightmost children.

```
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 $\mathrm{DFW}(0)$, yet are not all reachable in $\mathrm{DF}(K)$, for any $K \in \mathbb{N}$.

In other words, before proceeding past wait statements, the current round of all created subtasks, waited-for or not, are executed. Technically, at each step the label of at most one task can change status from R to W, though multiple labels can change status from W to R. Furthermore, DFW(0) is deterministic.

As the following result demonstrates, the $\mathrm{DFW}(K)$ scheduler is strictly more expressive than $\mathrm{DF}(K)$, in the sense that every global variable valuation that can be reached with $\mathrm{DF}(K)$ can also be reached with $\mathrm{DFW}(K)$, for all $K \in \mathbb{N}$, and that for every $K_0 \in \mathbb{N}$, there are programs whose set of valuations reached under $\mathrm{DFW}(K_0)$ cannot be reached by $\mathrm{DF}(K)$ 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 $\mathrm{DF}(K)$ 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 $\mathrm{DF}(K)$ 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 $\mathrm{DFW}(K)$ scheduler as the executions of a sequential program, we follow Bouajjani et al.'s intuition of *compositional* executions [4] 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

[4]We say i is *waiting for* j in configuration $\langle g, m \rangle$ when $\langle i, T[x := \textbf{wait } e], \_ \rangle \in m$ and $e(g, T) = j$.

bounding strategies for limiting concurrent behaviors to facilitate efficient program analysis, including context bounding [5, 2] and delay bounding [1]. We specialize Bouajjani et al.'s notion of compositional execution in order to fix a tight correspondence with 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 $k \in \mathbb{N}$, 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 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

Section 4's compositional semantics gives an alternate way to execute programs according to DFW(K), using nondeterministic choice (in the instantiation of fresh task interfaces): rather than storing tasks for later execution, we simply guess the global states that each task encounters at the beginning of its (up to K+1) rounds, to obtain one possible (K+1)-length interface before resuming its caller. In essence, querying a task for its interface at the point where it is called mimics the same control flow as a procedure call. 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, we assume 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.

```
// translation of var g: T
var G[K+1], Guess[K+1], Next[K+1]: T
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' // i.e. the translation of s

// translation of proc main() s
proc main()
  const Init[K+1]: T := G;
  var k: int := 0;
  delays := 0;
  Next := Guess := ⋆;
  s'; // i.e. the translation of 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 $\Sigma(P, K)$.

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

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

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

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

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

Figure 5: The compositional program semantics.

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 $\mathrm{DF}(K)$ 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 $\mathtt{Next}[K](c)$, i.e., the valuation of the `Next` vector's last element in c; then we define $R_\varphi(P) = \{\varphi(c) : c_0 \to \ldots \to c \text{ is finalized}\}$. Given this projection, we can show that the projected reachable global values in the K-delay sequentialization $\Sigma(P, K)$ of an asynchronous program P are precisely equal to the values reachable in P in the K-bounded compositional semantics.

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

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

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

6. COMPLEXITY

While Section 3 establishes that $\mathrm{DFW}(K)$ generally reaches more program variable valuations than $\mathrm{DF}(K)$ does, an obvious concern would be the cost at which it does so. In this section we demonstrate that despite the increased power of $\mathrm{DFW}(K)$ 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 $\mathrm{DF}(K)$. 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 $\mathrm{DFW}(K)$ *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 $\mathrm{DF}(K)$'s reachability problem [1], since $R(P, \mathrm{DF}(K)) = R(P, \mathrm{DFW}(K))$ for programs P without **wait** statements.

Lemma 3. $\mathrm{DFW}(K)$ *reachability is NP-hard.*

Our proof of NP-membership reduces the problem to 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. $\mathrm{DFW}(K)$ *reachability is in NP.*

Combining lemmas, we have a tight complexity result.

Theorem 3. $\mathrm{DFW}(K)$ *reachability is NP-complete.*

7. EMPIRICAL EVALUATION

We evaluate our $\mathrm{DFW}(K)$ scheduler empirically by comparing its sequentialization with an analogously implemented sequentialization of Emmi et al.'s $\mathrm{DF}(K)$ scheduler [1]; we have implemented both sequentializations in the `c2s` tool[5]. Since the $\mathrm{DF}(K)$ scheduler does not interpret **wait** statements, we pre-process each program given to the $\mathrm{DF}(K)$-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 $\mathrm{DF}(K)$'s or $\mathrm{DFW}(K)$'s) on a Boogie code

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

```
// 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.

representation[6] of the input asynchronous program, which is fed to the Corral verification engine [3] to detect whether an assertion violation can be reached within a given delay bound K. Our experiments were performed on a typical laptop (Macbook Pro 2013), and we report single-run times. We expect little variation in the comparison between DF and DFW across different hardware configurations, and have observed very little variation in runtime across multiple runs.

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[7] would be roughly equivalent to an automatic translation. Lacking an automatic translation from *asynchronous* C# programs, our experiments are tedious to carry out, and are thus limited to a few examples. We note that for programs without wait statements, the verification conditions ultimately generated by both sequentializations are quite similar, and the difference in solving them is insignificant. Experiments from existing works on sequentialization (e.g. by Emmi et al. [1]) do not consider programs with wait statements, and are therefore irrelevant to our current study.

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

[6]Boogie is an intermediate verification language [6].
[7]`https://github.com/smackers/smack`.

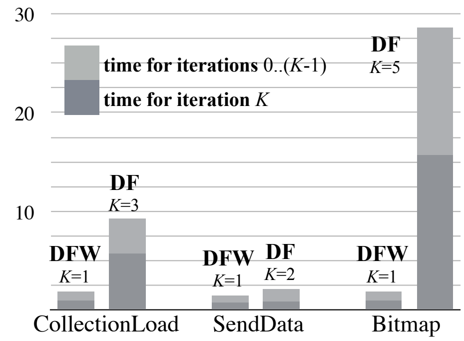

Figure 8: Time to bug detection (in seconds for three examples using the DFW and DF sequentializations. Each bar represents the aggregate time over increasing delay bounds, starting from zero, whereas the dark part indicates time spent for the smallest successful delay bound (K).

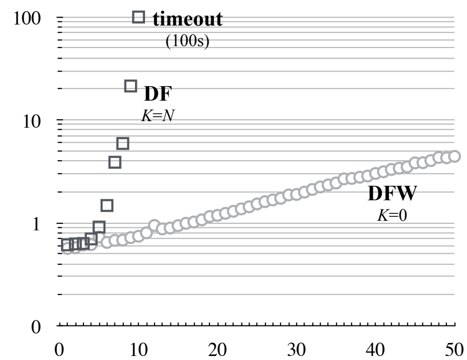

Figure 9: Time to bug detection (in seconds) for the parameterized example with N (on the X-axis) chained asynchronous calls. While the DFW sequentialization consistently discovers the bug without delays, DF requires $K = N$ delays, and times out at 100s for $N = 10$.

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 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 [5, 2]. 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. [4] 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 are based on a parameterized scheduler which can reduce exploration cost by taking into account program synchronization.

While Emmi et al.'s work [1] is indeed the starting point for our work, and the syntactic difference between our sequentializations is small, we believe our contribution is significant for the following reasons:

First, and more technically, besides the statements appearing in the translation of the wait statement, our DFW sequentialization must generalize DF. Our translation must repeatedly make guesses — once at each encountered wait statement — for the global state at which begins the sequence of asynchronous tasks called until the next-encountered wait statement (which must be equal to the global state reached by the next-encountered wait statement). In the case of DF, the global state at which the sequence of *all* asynchronous tasks begin is fixed once and for all (and must be equal to the global state reached by main). This extension is subtle, yet crucial.

Second, it is challenging to design a translation which

(A) correctly preserves causal information flow in the original program, while

(B) ensuring that the concurrent executions simulated by our sequential program never block because of wait statements.

While the relatively "easy" alternative translation listed in Figure 7 does satisfy A, it fails to satisfy B. Our formal development of the DFW sequentialization is a principled way to design a translation which satisfies both Properties A and B: we show that the executions admitted by the DFW scheduler (satisfying B) coincide exactly with our compositional semantics (satisfying A), bridging the gap between any given asynchronous program and its sequentialization.

Finally, comparing to approaches based on dynamic program exploration, while delay-bounding using techniques such as Chess [16] could capture the same sets of concurrent interleavings for a given delay bound, our static approach promises higher coverage: by using SMT-based symbolic reasoning engines, we can reason about many possible program data values at once, whereas dynamic techniques consider single concrete values. In practice this could allow us to catch data-dependent bugs undetected by a given dynamic technique.

9. ACKNOWLEDGEMENTS

We thank Ahmed Bouajjani for his participation in formative discussions.

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

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

[4] 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

[5] 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

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