Static Deadlock Detection for Asynchronous C# Programs

Anirudh Santhiar   Aditya Kanade

Indian Institute of Science, India

{anirudh_s,kanade}@csa.iisc.ernet.in

Abstract

Asynchronous programming is a standard approach for designing responsive applications. Modern languages such as C♯ provide async/await primitives for the disciplined use of asynchrony. In spite of this, programs can deadlock because of incorrect use of blocking operations along with non-blocking (asynchronous) operations. While developers are aware of this problem, there is no automated technique to detect deadlocks in asynchronous programs.

We present a novel representation of control flow and scheduling of asynchronous programs, called *continuation scheduling graph* and formulate necessary conditions for a deadlock to occur in a program. We design static analyses to construct continuation scheduling graphs of asynchronous C♯ programs and to identify deadlocks in them.

We have implemented the static analyses in a tool called DeadWait. Using DeadWait, we found 43 previously unknown deadlocks in 11 asynchronous C♯ libraries. We reported the deadlocks to the library developers. They have confirmed and fixed 40 of them.

CCS Concepts • **Software and its engineering → Deadlocks**; **Automated static analysis**

Keywords Asynchronous programs, async/await, C#, concurrency, static deadlock detection

1. Introduction

Asynchronous programming provides a means to defer high latency steps in a computation. Unlike a synchronous procedure call, an asynchronously called procedure does not run to completion. Rather, it returns the control to its caller immediately. Later, when the callee finishes execution, a callback procedure registered by the caller is invoked. This helps improve responsiveness of the application. The asynchronous style of programming is therefore used in many domains, ranging from device drivers and UI-driven desktop software to mobile, web and cloud applications.

The early models of asynchronous programming required developers to explicitly register callbacks with the asynchronous procedure calls. However, programming using callbacks becomes tedious because the program logic must be split manually by the developer into different callbacks, and the callbacks must be chained carefully. This problem is commonly known as "callback hell".

To overcome this problem, C♯ 5.0 [24] introduced the async/await primitives. These primitives allow the developer to write code in the familiar sequential style without explicit callbacks. An asynchronous procedure is declared with the async keyword. When called, it returns a *task* object and its caller can "await" the task. Awaiting suspends the execution of the caller, but does not block the thread it is running on. The code following the await instruction is the continuation that is automatically called back when the result of the

```
async Task<byte[]> GetContentsAsync(string url)
{
  var content = new MemoryStream();
  // Initialize an HttpWebRequest for the current url
  var webReq = (HttpWebRequest)WebRequest.Create(url);
  // Send request to url, await response
  Task<...> wTask = webReq.GetResponseAsync();
  await wTask;                                          C1
  using (WebResponse response = wTask.Result)
  { // Get data stream from response
    using (Stream rS = response.GetResponseStream())
    { // Copy the bytes in rS to content
      Task copyTask = rS.CopyToAsync(content);
      await copyTask.ConfigureAwait(false);
    }                                                   C2
  }
  return content.ToArray();
}

static int SumPageSizes()
{
  List<string> urlList = SetUpURLS(); int total = 0;
  foreach (var url in urlList){
    var cTask = GetContentsAsync(url);
    cTask.Wait(); byte[] urlContents = cTask.Result;
    total += urlContents.Length;
  }
  return total;
}
```

Figure 1: An asynchronous procedure GetContentsAsync to obtain contents of a URL as a byte array, and its synchronous client, SumPageSizes, that deadlocks.

callee is ready. Due to its simplicity, this paradigm is becoming the preferred mechanism for writing asynchronous code across programming languages. Apart from C$^\sharp$, C++ [39], Dart [26], ECMAScript [13], F$^\sharp$ [61], PHP [16], Python [59] and Scala [40] either support or plan to support `async`/`await` primitives. All major web browsers have now added support for `async`/`await` based JavaScript programs.

In addition to the non-blocking `await`s, C$^\sharp$ and other languages like C++, PHP and Scala also support *blocking or synchronous waits* on the task objects. Using a blocking wait, a caller can synchronize with an asynchronous callee. This mix of blocking and non-blocking operations gives rise to intricate scheduling dependencies and can even result in deadlocks. Developers are aware of the problem of deadlocks in asynchronous programs (*e.g.*, [63]), but there is no automated technique to detect them.

As an example, refer to the asynchronous C$^\sharp$ program in Figure 1, adapted from a tutorial [8]. It illustrates the use of asynchronous web requests. Given a URL `url`, the procedure `GetContentsAsync` calls an asynchronous procedure `GetResponseAsync` at line 7 to retrieve contents at `url`. `GetResponseAsync` eventually returns the contents, but to avoid blocking its caller, it returns a task object `wTask` immediately. The `GetContentsAsync` procedure awaits `wTask` at line 8. Lines 9–18, enclosed in the box $\mathcal{C}_1$, form the continuation that is invoked once the result encapsulated by `wTask` is ready. The compiler automatically extracts and schedules the continuation, freeing the developer from the burden of explicitly defining and registering a callback.

Once the contents at `url` are received, `GetContentsAsync` calls another asynchronous procedure `CopyToAsync` to copy the `content` stream to a byte array and awaits the task object `copyTask` returned by it at line 14. The corresponding continuation is shown inside the box $\mathcal{C}_2$. The procedure `SumPageSizes` runs on the UI thread and calls `GetContentsAsync` for each URL in a list and sums up the number of bytes retrieved. It uses `Wait` at line 25 to synchronously wait on the task object `cTask` returned by `GetContentsAsync`. This blocks the UI thread.

When an asynchronous call completes, the default behavior in C$^\sharp$ is to schedule its continuation on the same thread which issued the asynchronous call. This ensures that the continuation does not make invalid cross thread accesses. Due to the convenience it offers, other languages which support `async`/`await` may also follow the same default scheduling scheme. In Figure 1, the UI thread is blocked by `SumPageSizes` by invoking a `Wait` operation at line 25. The signaling operation to unblock it is in the continuation $\mathcal{C}_2$ which will be scheduled only after $\mathcal{C}_1$ finishes. Following the default scheduling scheme, $\mathcal{C}_1$ is scheduled to run on the UI thread, but it is blocked by `SumPageSizes`. This results in a deadlock.

In this work, we present a novel representation of the mixed synchronous and asynchronous control flow in asynchronous programs called the *continuation scheduling graph* (CSG). A CSG represents procedures and continuations, control flow between them and the threads on which they may be scheduled. From the CSG, we construct a *deadlock detection graph* (DDG) which represents threads, the scheduling of blocking and signaling operations onto the threads, and inter-dependence of blocking and signaling operations on each other. A cycle in the DDG indicates possibility of a deadlock. As a large number of C$^\sharp$ programs make extensive use of `async`/`await`, in this work, we focus on deadlock analysis of asynchronous C$^\sharp$ programs. The basic concepts and algorithms presented in this paper should be useful for other languages and for detection of programming errors other than deadlocks.

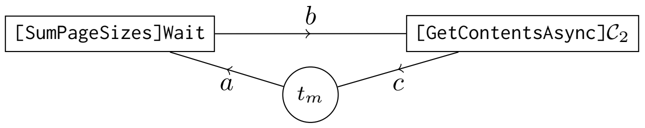

Figure 2: A partial rendering of the deadlock detection graph for the program in Figure 1.

Figure 2 shows the deadlock detection graph for the program in Figure 1. Only the nodes/edges that are relevant to the example are shown. The node labeled t_m stands for the main/UI thread. The edge a means that t_m may block due to the `Wait` operation in `SumPageSizes`. The edge b shows that the `Wait` operation can be signaled by the continuation $\mathcal{C}_2$ of `GetContentsAsync`. Finally, the edge c means that the continuation $\mathcal{C}_2$ can execute only if a continuation that can schedule it (the continuation $\mathcal{C}_1$ in this case) can be executed on t_m. The three edges in Figure 2 form a cycle which correctly identifies the deadlock.

This paper presents several new static analyses to achieve the objective of deadlock detection for asynchronous C$^\sharp$ programs. Constructing a CSG requires explicitly representing all the continuations and calling order between them, and determining the threads a continuation could run on. We extend the operational semantics for asynchronous C$^\sharp$ programs given by Bierman et al. [24], to track the scheduling of continuations onto threads, and present an algorithm that solves set-based constraints to infer the thread assignments. To construct the DDG, we identify correspondences between blocking and signaling operations, and how their scheduling depends on various threads and continuations.

We have implemented our algorithms in a tool called DeadWait and have used it to detect deadlocks in 11 asynchronous C$^\sharp$ libraries. These include libraries for asynchronous messaging [5] and cloud data management [12]. DeadWait found 43 previously unknown deadlocks in them. Clients of these libraries may deadlock simply by invoking some library procedures. These are serious bugs that could render the client applications unusable. We reported all the bugs to the library developers, and they have confirmed and fixed 40. Thus, our technique is useful in practice to detect real deadlocks that developers consider worth fixing.

The highlights of this work are as follows:

- We study the semantics and scheduling behavior of `async`/`await` based asynchronous programs, and the problem of deadlocks in them.
- We present CSGs, a novel program representation to capture control flow and scheduling of asynchronous programs. The CSG is an extension of the call graphs and can serve as a useful representation for other analysis techniques for asynchronous programs.
- Based on the CSGs, we provide the first deadlock detection technique for asynchronous C$^\sharp$ programs.
- The technique is implemented in a tool and evaluated on 11 libraries. It is effective in practice and found 43 new deadlocks. Developers fixed 40 of them.

2. Program Model

C$^\sharp$ is a full-featured object-oriented language. Bierman et al. give an operational semantics for Asynchronous C$^\sharp$ [24]. We extend the semantics to track the thread on which continuations are scheduled, and use it to define and detect deadlocks. In the interest of space, a formal treatment of our extensions is available in the supplementary document [58], and we provide an informal description here. We focus the discussion on C$^\sharp$'s asynchrony primitives, but the concepts also apply to other languages that support `async`/`await`.

2.1 Asynchrony Primitives

C$^\sharp$ supports asynchrony through several syntactic extensions to its grammar. First, procedures can be declared asynchronous using the `async` modifier, as shown in the syntactic class pd in Figure 3. In the production, m refers to a procedure name, and $\overline{\alpha\ x}$ to a list of parameters x and their types α. This is followed by the procedure body with declarations of the locals $\overline{\beta\ y}$ and the statements $\bar{s}$. The return type `Task<`δ`>` of an asynchronous procedure is used to encapsulate the *future result* of the procedure. We assume a return type of `Task<`δ`>` for ease of presentation. Return types `Task` and `void` are also permitted, and handled by our implementation. A *task* is an object of type `Task`. The task returned by an asynchronous procedure is initially in an incomplete state. It is marked complete when the procedure either fails (throwing an exception) or finishes successfully (producing a value of type δ), and does not change the state subsequently. When a task is complete, the exception/value is saved in the `Result` field of the task.

In Figure 1, `GetContentsAsync` intends to return a byte array. Since it is an asynchronous procedure, it actually returns an object of type `Task<byte[]>` as seen in the procedure declaration. The byte array can be read off using the `Result` field of this object when the asynchronous call finishes execution. This is what the caller `SumPageSizes` of

pd	$::=$	`async Task<`δ`>` $m(\overline{\alpha\ x})\ \{\overline{\beta\ y}; \bar{s}\}$	(Proc. decl.)
ae	$::=$	`await t`	(Await expression)
te	$::=$	`t.get_Result()` $\mid$ `t.Wait()`	(Blocking proc.)
	$\mid$	`t.ConfigureAwait(x)`	(Modify scheduling)
lte	$::=$	`t.SetResult(x)`	(Signaling proc.)
	$\mid$	`t.IsCompleted()`	(Check completion)
	$\mid$	`t.ContinueWith(x,b)`	(Register callback)

Figure 3: Syntactic extensions in C$^\sharp$ to support asynchrony.

`GetContentsAsync` does at line 25 on the task object `cTask` returned by `GetContentsAsync`.

The `await` operation is the second syntactic extension in C$^\sharp$ to facilitate asynchronous programming. A procedure can asynchronously wait for the completion of a task (returned to it by another procedure) using the `await` operation following the syntactic class ae in Figure 3. The `await` instructions may only be placed within asynchronous procedures, that is, procedures declared as `async`. If a procedure m has an `await t` statement, we say m *awaits* `t`.

C$^\sharp$ supports some expressions, identified by the syntactic class te in Figure 3, on task objects. For a task object `t`, `t.get_Result()` is equivalent to the field access `t.Result`. Another procedure in the syntactic class te is `t.Wait()`. Both of these are examples of procedures that block the calling thread until the task `t` completes. We refer to them as *blocking procedures*. Developers may exploit parallelism by scheduling a continuation that does not make invalid cross thread accesses on a thread-pool thread. They can do this by overriding the default scheduling scheme using `ConfigureAwait(false)`. Awaiting a task `t` as "`await t.ConfigureAwait(false)`" results in the continuations associated with the `await` being scheduled on a thread-pool thread. For example, invoking this procedure on `copyTask` at line 14 in Figure 1 results in scheduling of the continuation $\mathcal{C}_2$ in Figure 1 on a thread-pool thread.

Apart from these basic operations, developers use asynchronous procedures from the C$^\sharp$ libraries such as the asynchronous I/O procedures and task related procedures from the Task Parallel Library (TPL) [50]. We explain how we model them in Section 5.2. While developers predominantly use the `async`/`await` primitives at present, the program representations and analyses we propose in this paper may also be useful for legacy code that uses earlier asynchronous programming models [1, 2].

2.2 Explicating Asynchronous Control Flow

We present certain low-level primitives, given by the syntactic class lte in Figure 3, over task objects and then use them to explicate control flow of asynchronous C$^\sharp$ programs.

Low-level Primitives The completion state of a task cannot be modified directly. Instead, C$^\sharp$ provides wrappers around

[R-First]

{S← 1; L← Label(S);}

s ⇒ $L: s

[R-Return]

```
return v;
```

$\Downarrow$

```
1 task.SetResult(v);
2 return;
```

[R-Await-P]

{S++; L← Label(S);}

```
x = await t;
```

$\Downarrow$

```
1 if(!t.IsCompleted()){
2   state = $S;
3   t.ContinueWith(m̂,true);
4   return;
5 }
6 $L: x = t.get_Result();
```

[R-Await-D]

{S++; L← Label(S);}

```
x = await t.ConfigureAwait(false);
```

$\Downarrow$

```
1 if(!t.IsCompleted()){
2   state = $S;
3   t.ContinueWith(m̂,false);
4   return;
5 }
6 $L: x = t.get_Result();
```

[R-Async-Proc]

{SC← switch (state){ case 1: goto Label(1); ... case S: goto Label(S);}}

async Task<δ> $m(\overline{\alpha\ x})$ $\{\overline{\beta\ y}; \overline{s}\}$

$\Downarrow$

```
1 class C_m { $\overline{\alpha\ x}$; $\overline{\beta\ y}$; int state; Task<δ> task; void m̂(){ $SC; $\overline{s}$; } };
2 async Task<δ> m($\overline{\alpha\ x}$){
3   C_m sm = new C_m(); sm.$\overline{x}$ = $\overline{x}$; sm.$\overline{y}$ = default($\overline{\beta}$); sm.task = new Task(); sm.state = 1;
4   sm.m̂(); return sm.task;
5 }
```

Figure 4: Rewrite rules to explicate control flow of asynchronous procedures.

Task with procedures to mark the wrapped task as complete. The type TaskCompletionSource and its SetResult procedure are examples of this. For simplicity, we will assume that the SetResult procedure is supplied by the Task type itself. We call SetResult a *signaling procedure*.

The procedure IsCompleted on tasks returns true iff the receiver task has been marked complete. Marking a task as complete results in (i) signaling of procedures blocked on it and (ii) scheduling of continuations of associated await instructions. The Task type supplies the ContinueWith procedure to register a continuation x to be run when the task completes. The second argument b of ContinueWith determines where the continuation will be scheduled. The value of false schedules x on a thread-pool thread, and true schedules x on the thread that calls t.ContinueWith(x,true).

Rewriting Async Procedures to State Machines The asynchrony primitives, async/await, are at a higher level of abstraction. In order to analyze control flow within an asynchronous procedure precisely, we provide syntactic rewrite rules to implement async/await using the low-level primitives discussed above. Our subsequent analyses operate on the rewritten code. The rewrite rules are shown in Figure 4. They mimic the C$^\sharp$ 5.0 compiler transformations, that are being adopted by C++ [39] and Scala [40]. We omit some details about exception handling for brevity.

For each asynchronous procedure m, we create a new class C_m with two special fields state and task, and a new procedure $\hat{m}$. The overall strategy is to move the code in m to $\hat{m}$, and organize $\hat{m}$ as a state machine. We call $\hat{m}$ a *state machine procedure*. The state of $\hat{m}$ is maintained in the field state and the field task refers to the task object that m returns to its caller. An await (with corresponding continuation $\mathcal{C}$) is rewritten to cause $\hat{m}$ to update its state, and then register $\hat{m}$ itself as a callback. When the callback is invoked, $\hat{m}$ jumps to the state corresponding to continuation $\mathcal{C}$ and executes the code corresponding to $\mathcal{C}$.

The rewrite rules in Figure 4 are applied to the parse tree of m in a bottom-up and left-to-right manner. A rewrite rule R is of the form $\{\mathtt{pre}\}A \Rightarrow B$. It is applied if the statement rooted at a subtree matches pattern A. The subtree is then rewritten to statements matching pattern B. pre is a sequence of actions to be executed before performing the rewrite. If a rule does not require any actions, we simply use $A \Rightarrow B$. The actions are defined over the following action variables: S tracks the next state of the state machine and L is a label to identify different code locations that can be reached asynchronously. The function Label maps a state to a unique label. The action variables can be referred to in B by prefixing them with \$ sign.

The rule [R-First] is applied to the first statement s in m. It initializes the action variable S and attaches a label L corresponding to the initial state to statement s. The rule [R-Return] rewrites a return statement return v to a statement task.SetResult(v) to signal completion of the task object task, followed by a return statement.

We use the rewrite rule [R-Await-P] to desugar the await operation. We assume that all occurrences of await are in statements of the form x = await t. Before triggering the rule, pre actions increment action variable S, and generate a fresh label L corresponding to it. If the task being awaited is already complete, we can simply retrieve the corresponding result by invoking the get_Result procedure on it. Therefore, the re-written code first tests for t's completion using IsCompleted (line 1), and retrieves its result if it is completed (line 6). Line 6 is assigned the label L because when the task t completes, the control must resume from L. If it is incomplete, the code at lines 2–4 should execute, assigning the value of S to state, and registering $\hat{m}$ as a callback using the `ContinueWith` procedure of `t`. The rule [R-Await-D] is similar, except that it encodes the semantics of `ConfigureAwait(false)` by calling `ContinueWith` with the second argument set to `false`. This results in the callback to $\hat{m}$ being scheduled on the thread-pool.

Finally, the rule [R-Async-Proc] generates the class $\texttt{C}_m$ and the state machine procedure $\hat{m}$ for the asynchronous procedure m. Since $\hat{m}$ will be invoked repeatedly as a callback, all the parameters $\overline{\alpha\ x}$ and locals $\overline{\beta\ y}$ of m are lifted as fields of $\texttt{C}_m$ as shown at line 1. In addition, the fields `state` and `task` described earlier are also declared. The state machine procedure $\hat{m}$ contains the switch-case statement, referred to as `$SC`, followed by the statements $\bar{\texttt{s}}$ obtained from m by applying the rewrite rules. The switch-case statement `SC` is generated using the mapping between `S` and the associated labels in the action part of the rule. The asynchronous procedure m is rewritten to create an instance `sm` of $\texttt{C}_m$ and initialize the fields corresponding to the input arguments $\overline{\alpha\ x}$ and locals $\overline{\beta\ x}$ (line 3). It also creates the task object `sm.task` and initializes the state `sm.state` of the state machine object to `1`. It then synchronously invokes $\texttt{sm.}\hat{m}$ and returns the task object `sm.task` to its caller.

Let us examine how the rewritten code captures m's control flow correctly. The first time an asynchronous procedure is invoked, execution proceeds *synchronously* until the first `await` and then returns a fresh incomplete task to its caller. Therefore, notice that the original body of m is replaced by code to allocate a fresh `C_`m object `sm`, initialize its fields, and invoke $\texttt{sm.}\hat{m}$ synchronously. In turn, $\texttt{sm.}\hat{m}$ will execute from m's first statement until it reaches the first (desugared) `await t`. If the task `t` is not yet complete, it will update `sm.state`, register itself as a continuation with `t`, and return (lines 2–4 of [R-Await-P]). Once $\texttt{sm.}\hat{m}$ returns, m will return a task object `sm.task`, on which the caller can wait-/await. The next time $\texttt{sm.}\hat{m}$ is called-back on completion of `t`, the switch-case statement transfers control to the correct location to execute the required continuation. Eventually, when a callback to $\texttt{sm.}\hat{m}$ reaches the code generated by [R-Return], the result `v` will be stored in `sm.task` using the signaling procedure `SetResult`. This results in unblocking of wait operations and scheduling of continuations associated with await operations on `sm.task`.

2.3 Scheduling Scheme and Deadlocks

The thread on which a callback resumes depends on whether the `await` it corresponds to is *configured*. In the default case, the callback is scheduled on the same thread as the thread that registered it using the `ContinueWith` method. Alternatively, calling `ConfigureAwait(false)` on a task, as is done at line 14 in Figure 1 for `copyTask`, results in application of [R-Await-D] in Figure 4. The rule generates $\texttt{t.ContinueWith(}\hat{m}\texttt{,false)}$, that schedules the callback $\hat{m}$ on a thread-pool thread. A few other variants to determine the scheduling behavior are possible but it is beyond the scope of this paper to delve into them.

Real-world applications are large, and contain a maze of blocking and non-blocking (asynchronous) operations. It is non-trivial to get a clear view of the underlying scheduling choices, raising the possibility of deadlocks. A set of threads D is said to *deadlock* when each thread in D is blocked and the operation it depends on for unblocking may only run in the future on a blocked thread in D.

3. Continuation Scheduling Graph

A call graph is a fundamental data structure used by many types of program analyses. The procedures in a program are the nodes of the call graph and the synchronous control flow between them is represented by directed edges. For an accurate representation of control flow in asynchronous programs, we also need to represent continuations and their scheduling by signaling procedures. Therefore, we introduce a novel representation for asynchronous programs, called the continuation scheduling graph.

3.1 State Machine Model of Asynchronous Procedures

As discussed in Section 2.2, an asynchronous procedure m containing `await` statements is re-written into a state machine procedure $\hat{m}$ that does not contain `await`. Let $\mathcal{V}_{\hat{m}}$ be the set of variables of $\hat{m}$ with $\texttt{state} \in \mathcal{V}_{\hat{m}}$ as a distinguished variable representing the discrete states that it can be in. When a continuation is registered, by default, the current scheduling context is *preserved*. That is, the continuation will run on the same thread it is registered from. An exception is when the thread is a thread-pool thread, in which case, the continuation may run on an arbitrary thread from the thread-pool. The developer can choose to *drop* the context using `ConfigureAwait(false)`. We represent the two cases with labels ‘p’ and ‘d’ respectively. Let $\mathcal{L} = \{\mathrm{p}, \mathrm{d}\}$.

DEFINITION 1. *The state machine model of an asynchronous procedure m is a tuple $(\mathcal{S}_{\hat{m}}, init_{\hat{m}}, \texttt{State}_{\hat{m}}, \Delta_{\hat{m}})$, where*

- *$\mathcal{S}_{\hat{m}}$ is a finite set of states representing the discrete values that the variable $\texttt{state} \in \mathcal{V}_{\hat{m}}$ can take.*
- *$init_{\hat{m}} \in \mathcal{S}_{\hat{m}}$ denotes the initial state.*
- *$\texttt{State}_{\hat{m}} : \mathcal{I}_{\hat{m}} \to \mathbb{P}(\mathcal{S}_{\hat{m}})$ maps each instruction $i \in \mathcal{I}_{\hat{m}}$ in $\hat{m}$ to the set of states it belongs to.*
- *$\Delta_{\hat{m}} \subseteq \mathcal{S}_{\hat{m}} \times \mathcal{S}_{\hat{m}} \times \mathcal{L} \times \mathcal{V}_{\hat{m}}$ denotes the labeled transition relation of the state machine. $(s, s', \mathrm{p}, v) \in \Delta_{\hat{m}}$ if s registers a continuation corresponding to code in the state s' on a task v, requiring it to schedule the continuation on the same thread as s up on completion of v. Analogously, $(s, s', \mathrm{d}, v) \in \Delta_{\hat{m}}$ but the continuation corresponding to s' is to be scheduled on a thread-pool thread. If s is scheduled on the thread-pool then s' is scheduled on an arbitrary thread of the thread-pool, irrespective of the transition label. In either case, if the task v is already complete at the time of registering the continuation state s' then the control flows sequentially from s to s'.*

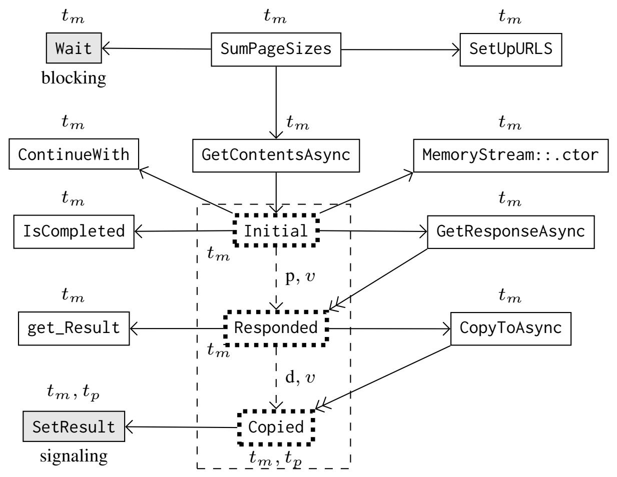

Figure 5: The continuation scheduling graph for the program in Figure 1. Not all nodes and edges are shown.

The `await` statements may occur within conditionals and loops in the source code. This results in complicated control flow where an instruction may appear in more than one state. Our state machine model allows this. The map $\mathsf{State}_{\hat{m}}$ assigns a set of states to each instruction.

3.2 Control Flow of Asynchronous Programs

An asynchronous program P has synchronous as well as asynchronous procedures. We now present an analogue of call graphs for asynchronous programs called the continuation scheduling graph. In addition to the control flow, it also represents the threads that a procedure or continuation may get scheduled on. In the definition below, we use $\uplus$ for disjoint union. Let $\mathcal{V}_P$ be the set of variables of state machine procedures corresponding to all asynchronous procedures in P. Let $\mathcal{T}$ be a finite set of thread-ids which abstractly represents the threads used in P. Let $\mathcal{L} = \{\mathrm{p}, \mathrm{d}\}$ as before.

DEFINITION 2. *The continuation scheduling graph (CSG) for an asynchronous program P is a directed graph $G = (N, E, \zeta)$. The set of nodes is $N = N_1 \uplus N_2$ where N_1 represents synchronous procedures in P, and N_2 is the union of the set of states of the state machine models of asynchronous procedures in P. We assume that the states of any two state machine procedures are distinct from each other. The set of edges is $E = E_1 \uplus E_2 \uplus E_3$ where*

- *$E_1 \subseteq N \times N$ is the set of synchronous call edges. These edges are depicted using $\rightarrow$.*
- *$E_2 \subseteq N \times N_2$ is the set of callback edges. A callback edge connects a node $n \in N$ to a state $s \in N_2$ denoting a continuation c', if n contains a statement that may call back c'. These edges are depicted using $\twoheadrightarrow$.*
- *$E_3 \subseteq N_2 \times N_2 \times \mathcal{L} \times \mathcal{V}_P$ is the set of state machine transitions for all asynchronous procedures in P. These edges are depicted using $\dashrightarrow$.*

Finally, the map $\zeta : N \rightarrow \mathbb{P}(\mathcal{T})$ maps each node to the set of threads it may be scheduled on.

The state machine procedure of `GetContentsAsync` from Figure 1 consists of three states - one each for the initial code fragment, and continuations $\mathcal{C}_1$ and $\mathcal{C}_2$. Let us call them `Initial`, `Responded` and `Copied`. Figure 5 shows the CSG for the program in Figure 1. While the graph encodes all the information contained in a traditional call graph using the edges depicted by $\rightarrow$, we point out two major differences. First, there is no node corresponding to $\hat{m}$, the state machine procedure corresponding to `GetContentsAsync`; it is instead replaced by the corresponding state machine model (shown in the dashed rectangle). The incoming and outgoing edges of $\hat{m}$ are mapped to appropriate states of its state machine. For example, the incoming edge from `GetContentsAsync` goes to the `Initial` state. Second, the edges depicted by $\twoheadrightarrow$, absent in call graphs, encode asynchronous control flow. For example, the edge from `CopyToAsync` to the `Copied` state of `GetContentsAsync`'s state machine captures the fact that the latter will be scheduled when the former completes.

The nodes in Figure 5 are tagged with the threads on which they may run. Let us assume that the root node (`SumPageSizes` in Figure 5) runs on the main thread, t_m. All nodes reachable from it through synchronous call edges must run on the same thread. An asynchronously reachable node may resume on another thread at a later point, depending on the scheduling choice made. For example, the callback corresponding to the `Copied` state would resume on a thread-pool thread because the code that registered it directed the compiler to drop the context. Therefore, it is tagged using t_p, representing an arbitrary thread-pool thread. We also tag continuations with the thread that the predecessor state ran on, because control could flow synchronously in the case that the awaited task is already complete. Hence, we tag the `Copied` state with t_m as well.

4. Static Analysis

We describe the static analysis to detect deadlocks in asynchronous C$^\sharp$ programs in this section. We consider an asynchronous program P containing procedures $\mathcal{M}$ and instructions $\mathcal{I}$. We assume that P has a main procedure which is synchronous. Let $\mathcal{M}_a$ and $\mathcal{M}_{sm}$ respectively be the sets of asynchronous procedures and corresponding state machine procedures generated as per Section 2.2.

4.1 Call Graph and Points-to Relation

As the first step, we compute the call graph and may points-to information for the program P. At each call-site, the call graph identifies a static over-approximation of the set of target procedures that can be called at runtime. The desugaring of `async`/`await`, explained in Section 2.2, results in code which uses higher-order functions that take delegates as arguments. For example, in the rewrite rule [R-Await-P] in Figure 4, `ContinueWith` is a higher-order function that takes a delegate corresponding to $\hat{m}$ as the first argument. Madhavan et al. [53] present a modular heap analysis for C$^\sharp$ programs that handles higher-order functions and delegates. It performs a bottom-up computation of procedure summaries. To be sound in the presence of multi-threading and asynchronous procedures, we use the flow-insensitive version of the analysis. We refer the reader to [53] for more details. Once the bottom-up analysis is complete, we perform a top-down analysis to compute the may points-to information.

For greater precision, we use a finite set of *call contexts* $\mathcal{C}$ and qualify local variables of procedures using them. To disambiguate abstract heap objects allocated at the same allocation site, we use a finite set of *heap contexts* $\mathcal{HC}$. Formally, a *call graph* is $\mathsf{CG} : \mathcal{I} \times \mathcal{M} \times \mathcal{C} \to \mathbb{P}(\mathcal{M} \times \mathcal{C})$. A tuple $\langle i, m, c, m', c' \rangle \in \mathsf{CG}$ says that in call context c, instruction i in procedure m can call procedure m' with call context c'. The *points-to relation* $\mathsf{PointsTo} : \mathcal{V} \times \mathcal{C} \to \mathbb{P}(\mathcal{H} \times \mathcal{HC})$ maps local variables $v \in \mathcal{V}$ in context $c \in \mathcal{C}$ to their may-point-to sets. An element in a may-points-to set is a pair (o, hc), with $o \in \mathcal{H}$ standing for an abstract heap object (allocation site), and $hc \in \mathcal{HC}$ for a heap context. We describe the configurations used for computing the call/heap context in our experiments in Section 5.1.

We use $[\,]$ to stand for the empty context, and _ for a don't care value. We define $\mathsf{MayAlias}(x, c, y, c') \equiv \exists o, hc.(o, hc) \in \mathsf{PointsTo}(x, c) \cap \mathsf{PointsTo}(y, c')$. The function Proc returns the procedure containing instruction i, and the function Receiver returns the receiver variable for the call instruction i. Let $\mathsf{This}(m)$ return the "`this`" variable for procedure m.

4.2 Extracting State Machines

Let $\hat{m} \in \mathcal{M}_{sm}$ be the state machine procedure for an asynchronous procedure $m \in \mathcal{M}_a$. We analyze $\hat{m}$ and extract the state machine model $(\mathcal{S}_{\hat{m}}, init_{\hat{m}}, \mathtt{State}_{\hat{m}}, \Delta_{\hat{m}})$ as defined in Definition 1. As described in Section 2.2, we use a distinguished, discrete-valued variable `state` to denote the state that $\hat{m}$ is in. The set of states $\mathcal{S}_{\hat{m}}$ is same as this set of discrete values. The initial state $init_{\hat{m}}$ is given by the value assigned to the `state` variable in the state machine initialization code in the corresponding asynchronous procedure (see, for example, line 3 in the rule [R-Async-Proc] in Figure 4). To determine instructions belonging to each state, we perform an intra-procedural, path-sensitive dataflow analysis. The set of dataflow facts is the powerset of the set of states $\mathcal{S}_{\hat{m}}$ of the state machine. The assignments to the `state` variable and conditions involving it, update the dataflow facts in a straightforward manner.

To faithfully model the asynchronous control flow, we give special treatment to the code fragment that checks whether the task to be awaited is already complete (using `IsCompleted`) and if not, registers a continuation as a callback (using `ContinueWith`). Line 1 of rule [R-Await-P] in Figure 4 is an example of such a code fragment. In the true branch, (i) the `state` variable is updated and then (ii) the continuation is registered. The instruction (i) must precede the instruction (ii) at runtime to ensure that a proper state is set before the callback is registered (and scheduled). For the purposes of analysis, we interchange the positions of statements (i) and (ii). This allows us to compute the right set of states to which instruction (ii) belongs. If the task to be awaited is already complete, the control falls through to the next state. For analysis, we explicitly add an else branch which updates the `state` variable to the successor state.

The transition relation $\Delta_{\hat{m}}$ between states is computed using the instruction-to-states map $\mathtt{State}_{\hat{m}}$. First, the analysis identifies the instructions that register continuations via the `ContinueWith` procedure. Let i be such an instruction which registers a continuation on a receiver variable v. As discussed in the previous paragraph, the instruction i will be immediately followed by an assignment j to the `state` variable to set the target state for the callback. Let A be the dataflow fact after j. The analysis adds the transitions $\{(s, s', l, v) : s \in \mathtt{State}_m(i) \wedge s' \in A\}$ to Δ_m. The label l can be determined directly by checking the second argument of `ContinueWith`, as discussed in Section 2.3.

4.3 CSG Construction

Since the call graph CG constructed in Section 4.1 does not model asynchronous control flow, we construct the continuation scheduling graph (CSG). Recall that a CSG is $G = (N_1 \uplus N_2, E_1 \uplus E_2 \uplus E_3, \zeta)$ as defined in Definition 2. Similar to the call graph, we augment nodes of the CSG with call contexts. Figure 6 gives the rules to construct the CSG. We explain them step-by-step below.

Embedding State Machines Rules R1-R4 are applied to each state machine procedure in each call context it appears in. Let $(\hat{m}, c)$ be a state machine procedure in a call context c. Let $\langle i, m', c', \hat{m}, c \rangle$ be an incoming edge to it. The rule R1 (in Figure 6) adds a synchronous call edge from (m', c') to $(init_{\hat{m}}, c)$ to the set E_1 of the CSG. Let $\langle i, \hat{m}, c, m', c' \rangle$ be an outgoing edge from $(\hat{m}, c)$ such that the call instruction i belongs to a state s of the state machine of $\hat{m}$. The rule R2 adds a synchronous call edge from (s, c) to (m', c') to the set E_1. The rule R3 adds an edge from (s, c) to (s', c) tagged with a label l and variable v to the set E_3 of the CSG if (s, s', l, v) is a transition in the state machine of $\hat{m}$.

The callback edges are represented by the set E_2 of the CSG. An edge (s, s', l, v) in the state machine of $\hat{m}$ means that s has registered a callback on a task object `t` pointed to by variable v and the callback will schedule the continuation represented by the state s' according to the label l. In order to identify callback edges coming into (s', c), we find the signaling procedures that may signal completion of the task `t`. Let i be a call to a signaling procedure such that the receiver object $\mathsf{Receiver}(i)$ in a context c' may alias with (v, c). Let $\hat{m}'$ and s'' be the state machine procedure and its state to which i belongs. The rule R4 adds a callback edge from (s'', c') to (s', c) to the set E_2.

(CSG edges)

$$\frac{\hat{m} \in \mathcal{M}_{sm} \qquad init_{\hat{m}} \in \mathsf{S}_{\hat{m}} \qquad \langle i, m', c', \hat{m}, c \rangle \in \mathsf{CG}}{\langle (m', c'), (init_{\hat{m}}, c) \rangle \in E_1}\text{[R1]} \qquad \frac{\hat{m} \in \mathcal{M}_{sm} \qquad s \in \mathsf{State}_{\hat{m}}(i) \qquad \langle i, \hat{m}, c, m', c' \rangle \in \mathsf{CG}}{\langle (s, c), (m', c') \rangle \in E_1}\text{[R2]} \qquad \frac{\hat{m} \in \mathcal{M}_{sm} \qquad (s, s', l, v) \in \Delta_{\hat{m}} \qquad \langle \_, \_, \_, \hat{m}, c \rangle \in \mathsf{CG}}{\langle (s, c), (s', c), l, v \rangle \in E_3}\text{[R3]}$$

$$\frac{\hat{m} \in \mathcal{M}_{sm} \qquad (s, s', l, v) \in \Delta_{\hat{m}} \qquad i \in \mathcal{I} \text{ is a call to a signaling procedure} \qquad \mathsf{MayAlias}(v, c, \mathsf{Receiver}(i), c') \qquad \mathsf{Proc}(i) = \hat{m}' \in \mathcal{M}_{sm} \qquad s'' \in \mathsf{State}_{\hat{m}'}(i)}{\langle (s'', c'), (s', c) \rangle \in E_2}\text{[R4]}$$

(Map from nodes to threads)

$$\forall (m_{start}, c) \in N, \text{ initialize } \eta((m_{start}, c)) \text{ with fresh thread-id} \quad (1) \qquad (m, c) \to (m', c') \implies \eta((m', c')) \supseteq \eta((m, c)) \quad (2)$$

$$(s, c) \xrightarrow{\mathrm{p}, v} (s', c) \implies \eta((s', c)) \supseteq \eta((s, c)) \quad (3) \qquad (s, c) \xrightarrow{\mathrm{d}, v} (s', c) \implies \eta(s', c) \supseteq \eta((s, c)) \cup \{t_p\} \quad (4)$$

Figure 6: Rules to construct the continuation scheduling graphs.

(Reachability)

$$(m_1, c_1) \rightsquigarrow^0 (m_1, c_1) \qquad (m_1, c_1) \rightsquigarrow^{n+1} (m_2, c_2) \equiv \exists m, c : (m_1, c_1) \rightsquigarrow^n (m, c) \wedge \langle (m, c), (m_2, c_2) \rangle \in E_1 \cup E_2$$

(DDG edges)

$$\text{[D1]}\ \frac{t_i \in \zeta((m_b, c_b))}{\langle t_i, (m_b, c_b) \rangle \in \mathcal{E}} \qquad \text{[D2]}\ \frac{\mathsf{MayAlias}(\mathsf{This}(m_s), c_s, \mathsf{This}(m_b), c_b)}{\langle (m_b, c_b), (m_s, c_s) \rangle \in \mathcal{E}} \qquad \text{[D3]}\ \frac{(s, c') \text{ is a continuation} \qquad (s, c') \rightsquigarrow^* (m_s, c_s) \qquad t_i \in \zeta((s, c'))}{\langle (m_s, c_s), t_i \rangle \in \mathcal{E}}$$

Figure 7: Rules to construct the deadlock detection graphs.

Mapping Nodes to Threads After removing unreachable nodes, let $N = N_1 \uplus N_2$ be the nodes of the CSG, and $\mathcal{T}$ be a finite set of thread-ids which abstractly represent the threads in the program. We now construct the map ζ which assigns the set of abstract threads to each node. If $\zeta(n) = A$ then n may be scheduled on the threads in A.

A thread is abstracted by the node in the CSG that represents its start method. We use m_{start} to denote methods that begin threads, such as `Thread::Start`. We set $\eta((Main, []))$ to $\{t_m\}$. Other nodes with thread start methods are tagged using a fresh thread identifier from $\{t_1, \dots t_k\}$ (Eq. 1, Figure 6) where k is the number of such nodes in the CSG. We abstract the thread-pool threads by a distinct thread-id t_p. Once this initialization is performed, the rules in Eq. (2) – Eq. (4) are applied iteratively until a fixpoint is reached. All synchronous callees (m', c') of a caller (m, c) will execute on the same thread as (m, c) (Eq. 2). The thread on which a callback (s', c) executes depends on how it was registered by its predecessor state (s, c); it can either be scheduled to preserve the original thread (Eq. 3) or to execute on the thread-pool (Eq. 4). In the latter case, the continuation could still execute on the original thread in the case that the awaited task was already complete when the `await` statement ran. Therefore, $\eta(s', c)$ must be a superset of the union of $\eta(s, c)$ and the singleton set $\{t_p\}$ denoting the thread-pool threads.

Soundness Using a sound points-to analysis, we track all call sites to signaling procedures whose receiver may alias with the receiver of a call to `ContinueWith` generated while desugaring an `await`. The rule R4 of Figure 6 then adds the required callback edges to the CSG. Thus, we do not miss asynchronous control flow arising out of suspending at an `await`. The synchronous calls are over-approximated using a sound call graph construction algorithm. Further, the algorithm in Section 4.2 identifies the state machine transitions soundly. By induction on the length of paths from a root of the CSG to a node n, we can show that ζ identifies the set of threads that n may be scheduled on.

4.4 Deadlock Detection

To detect deadlocks in an asynchronous program, we identify the scheduling dependencies between blocking and signaling procedures of the program. We define the deadlock detection graph for this purpose.

DEFINITION 3. *The deadlock detection graph (DDG) for an asynchronous program P is a directed graph $\mathcal{G} = (\mathcal{X}, \mathcal{E})$. The nodes $\mathcal{X}$ represent the abstract thread-ids, and blocking and signaling procedures in the CSG of P. An edge in $\mathcal{E}$ is of one of the following forms:*

- *An edge from a thread t_i to a blocking procedure (m_b, c_b) indicating that t_i may be blocked on (m_b, c_b).*
- *An edge from a blocking procedure (m_b, c_b) to a signaling procedure (m_s, c_s) indicating that (m_b, c_b) may be signaled by (m_s, c_s).*
- *An edge from a signaling procedure (m_s, c_s) to a thread t_i indicating that (m_s, c_s) may be scheduled on t_i or depends on a continuation that may be scheduled on t_i.*

Our analysis constructs the DDG using the CSG and points-to information. The edges of the DDG are computed as the smallest set obtained by applying the rules in Figure 7. The rule D1 (in Figure 7) says that if a blocking method (m_b, c_b) could run on a thread t_i according to the node-to-threads map ζ of the CSG, then we add the edge $\langle t_i, (m_b, c_b)\rangle$ to the set $\mathcal{E}$ of the DDG. The rule D2 adds an edge from a node (m_b, c_b) representing a blocking procedure, to a node (m_s, c_s) representing a signaling procedure if their `this` variables may alias in the respective contexts.

We add an edge from a signaling procedure (m_s, c_s) to a thread t_i if the procedure may be scheduled on t_i. In general, there can be a chain of continuations that have to be scheduled to eventually schedule (m_s, c_s). If any of these continuations may run on t_i then also we add an edge from (m_s, c_s) to t_i to the DDG. This is captured in the rule D3. The reachability relation $\rightsquigarrow^*$ used in D3 is defined inductively in Figure 7. It is a transitive closure over the synchronous call edges E_1 and callback edges E_2 of the CSG. The rules to construct the DDG only add edges between a finite set of vertices, trivially ensuring termination. To detect deadlocks, we identify cycles [62] in the DDG and report the cycles found as deadlocks.

As an example, the `SetResult` procedure in Figure 5 is a signaling procedure which is called synchronously from the `Copied` state. The `Copied` state is reachable from the `Responded` state through a synchronous call to `CopyToAsync` and a callback from it. The `Responded` state may be scheduled on the main thread t_m. The `Responded` and `Copied` states are referred to as the continuations $\mathcal{C}_1$ and $\mathcal{C}_2$ in Section 1. In Figure 2, the edge c represents that $\mathcal{C}_2$ depends on t_m (due to the continuation $\mathcal{C}_1$). Since the `SetResult` procedure is introduced later in the paper, we use $\mathcal{C}_2$ instead of the signaling procedure `SetResult` in Figure 2. The other edges a and b are respectively generated by the rules D1 and D2. Since they form a cycle, a deadlock is reported.

Filtering to Avoid False Positives (i) The function ζ maps a node for a continuation (s, c') to the threads it may run on when scheduled asynchronously as a callback. In addition, it also maps it to the threads that it may run on when scheduled *synchronously*, if the task awaited is already complete. Let ζ_a be the map that considers only the former case. We use ζ_a in rule D3 of Figure 7 in place of ζ to construct the DDG. (ii) We use the thread-id t_p to abstractly represent an arbitrary number of thread-pool threads. Even though t_p is represented in the DDG, no cycle involving t_p is reported. (iii) Some calls to blocking procedures, such as the call to `get_Result` at line 6 in [R-Await-P], are generated by the compiler when it applies the rewrite rules of Figure 4. We do not report cycles involving them as deadlocks.

Soundness When a set D of threads of an asynchronous program deadlock, all of them must be blocked by some blocking procedures and the respective signaling procedures may only run in the future if a continuation is run on one of the threads in D. From the definition of the DDG, it follows that a cycle in the DDG is a necessary condition for a deadlock. The DDG nodes include all reachable blocking and signaling procedures, and abstractions of threads. Moreover, the dependencies between them are correctly captured. Our algorithm therefore finds all cycles symptomatic of possible deadlocks in an asynchronous program.

We now discuss the effect of the filtering techniques on soundness: (i) Let t_i be a thread such that $t_i \in \zeta((s, c')) \setminus \zeta_a((s, c'))$. Let (m_s, c_s) be a signaling procedure such that $(s, c') \rightsquigarrow^* (m_s, c_s)$. As (s, c') would already have executed synchronously on t_i, the scheduling of (m_s, c_s) does not depend on that of (s, c') and cannot be a cause for a deadlock. Our filtering technique therefore does not add an edge from (m_s, c_s) to t_i. (ii) In theory, deadlocks involving the abstract thread-pool thread t_p are possible and filtering such cycles may result in false negatives. However, in practice, they are unlikely because the thread-pool can grow dynamically in C$^\sharp$. (iii) Suppose *op* is a call to a blocking procedure inserted by the compiler. When *op* is reached synchronously, the task `t` must have completed in the predecessor state as confirmed by the conditional check `IsCompleted` on `t`. If *op* is reached asynchronously, then `t` must have completed for the corresponding continuation to be scheduled. In other words, by construction, the compiler-generated blocking operations and the matching signaling operations are related by a *must-happen-before relation*. Therefore, our third filtering mechanism is sound.

5. Implementation

5.1 Deadlock Detection Tool

We have implemented our analysis in a tool called DeadWait, written in C$^\sharp$ using the Microsoft Phoenix compiler framework. Our analysis requires a call graph and points-to relation as input. At present, we adapt an existing bottom-up heap analysis, Seal [53], to compute them. In a preprocessing pass, we renamed overloaded generic procedures in our benchmarks to overcome a limitation of Seal that it does not distinguish between generic overloads. Using Seal, we pre-computed summaries for a subset of namespaces exported by the .NET framework DLLs *mscorlib*, *System* and *System.Core*, that supply the functionality commonly used by C$^\sharp$ programs. These DLLs were analyzed once, and the summaries were reused for all our benchmarks. We then configured Seal to be flow-insensitive, and to use unbounded call-strings (that could be as long as the longest acyclic path in the call graph) as heap contexts, and obtained procedure summaries and a call graph for each benchmark program. We implemented a top-down pass within Seal to compute the may points-to relation. We used call-strings of length 1 as call contexts for procedures making indirect calls, for example via delegates, in the top-down pass. Owing to limitations of Seal, we currently analyze asynchronous C$^\sharp$ libraries exercised by single threaded clients.

Our deadlock reports consist of threads, and blocking and signaling procedures involved in cycles in the DDG. DeadWait can be configured to output the shortest CSG paths from the start procedures of threads to the blocking and signaling procedures involved in the deadlock as a debugging aid. DeadWait also supports visualization of CSGs.

5.2 Modeling of Framework Procedures

For our experiments, we considered `Task::get_Result` and `Task::Wait` as blocking procedures, and `SetResult` of `AsyncVoidMethodBuilder` and `AsyncTaskMethodBuilder` classes, along with their generic counterparts, as signaling procedures. None of our benchmarks used legacy asynchronous programming models [1, 2], and our implementation does not aim to handle them. To make our technique applicable to modern C$^\sharp$ code that uses `async/await`, we model asynchronous framework procedures, and some constructs from the Task Parallel Library (TPL) [50] as follows.

Asynchronous Framework Procedures The .NET framework exposes procedures for asynchronous I/O, network operations, etc. These are written carefully so that their continuations resume on thread-pool threads. Blocking on the result of these procedures could deadlock only if the thread-pool gets exhausted. We do not analyze these procedures, and use stubs that simply return fresh tasks. In the CSG, we model these procedures using state machines with a single state calling a signaling procedure on the returned task.

TPL Procedures The .NET framework exposes TPL to introduce parallelism into C$^\sharp$ programs. TPL exports the `Task` type used to represent future results of asynchronous procedures. We model `Task::Run` in a manner similar to our modeling of framework asynchronous procedures described earlier. We handle `Task::ContinueWith` using a rule similar to the rule R4 shown in Figure 6. Our modeling can be extended to TPL procedures such as `Task::WhenAll` that accept a list of tasks as an argument.

6. Evaluation

6.1 Experimental Results

Benchmark Selection and Clients Of the Microsoft Azure Github repositories [11] that used asynchrony primitives, and had blocking operations on tasks reachable via public APIs, we picked the three most popular as candidates for evaluation. We excluded sample and test code from consideration. The selected repositories are azure-sdk-for-net [7], autorest [6] and dotnetty [10]. The azure-sdk-for-net repository had more than 40 different projects. We picked projects dealing with authentication and cloud storage from among these. We also picked Microsoft Azure's amqpnetlite [5], a popular messaging protocol library that had over 100 stars on Github. We augmented these benchmarks with 6 other libraries spanning diverse domains from other organizations. These include Microsoft and Citrix libraries [3, 12, 15], a popular library to interface with Twitter [4], a commercial accounting software [9] and a number-tracking API [14].

Our focus was to find deadlocks in libraries as these can affect multiple client programs. For each library, we systematically wrote clients to reach APIs that called blocking procedures on the result of asynchronous procedures of the library. Our clients reached the blocking procedures, although they might not have reached every asynchronous procedure in the library. The clients did not require very precise initialization routines, or specific concrete values, and were easy to write. We note that analyzing existing client programs may reveal deadlocks in client code, but may miss some deadlocks in libraries unless they call the problematic APIs. Although we wrote clients for each library manually, they could potentially be constructed automatically [55].

We list some of the characteristics of our benchmarks in Table 1. The SLOC column shows the number of lines of source code (SLOC) analyzed in the bottom up pass of Seal. In three cases, Seal timed out on the full library (with a time-out of 60 min). We therefore restricted the analysis to only the namespaces imported by our clients. We indicate these partially analyzed benchmarks using "(p)" against the benchmark name. The SLOC counts are for only these namespaces. The other columns show the number of classes and procedures, and the number of asynchronous procedures analyzed. The sizes of the clients we wrote are listed under "Client SLOC". The one-time summary computation for the framework DLLs analyzed 4546 procedures, and their sizes are not counted as part of the SLOC column in Table 1.

Timing and Results We ran our experiments using a virtual machine running Windows 10, on an Intel i5 CPU with 4 cores clocked at 3.2GHz, with 10GB of RAM. Both Seal and DeadWait use only a single core, and at most 4GB of memory. Seal took a total of 27 minutes for the one-time analysis of framework DLLs. Computing the call graphs and points-to relations, followed by CSG and DDG construction, and deadlock detection and reporting took 20 min on average, taking at least 2 min and at most 57 min.

We summarize the results of our experiments in Table 2. The "Reports before filtering" column gives the number of deadlocks reported by DeadWait without using any filtering techniques (see Section 4.4). The "Reports after filtering" column indicates the number of deadlocks reported after applying the filtering techniques. In total, DeadWait reported 66 deadlocks for the 11 benchmarks. In the case of dinero, 22 deadlocks were reported by DeadWait, but they were symptomatic of only 3 underlying (real) bugs. Thus, DeadWait reported 47 unique potential bugs.

To confirm the bugs, we reproduced the deadlocks wherever possible. Some of our benchmarks were libraries to interact with cloud services, and we did not possess credentials to drive their clients. In such cases, we validated them by manual inspection. All the real deadlocks found were previously unknown. The "Bugs" column reports the number of

Table 1: Benchmarks analyzed for deadlocks.

Benchmark	Description	SLOC	Client SLOC	Classes	Procs.	Async Procs.
amqpnetlite (p)	Async messaging protocol	7620	43	84	733	16
authentication	Authentication module (azure-sdk-for-net)	8294	36	215	1249	50
autorest.core	RESTful web services generator	7617	48	266	1645	3
dinero	Commercial accounting program SDK	1593	157	110	533	11
dotnetty	Event-driven asynchronous network app framework	19 418	58	361	2635	4
hbase-sdk-for-net	Microsoft HBase client library to manage cloud data	2884	134	131	588	34
numerous-app	.NET API for number tracking app	1073	57	80	447	40
o365rwsclient	API for Microsoft Office reporting web service	1867	39	66	366	1
sharefile-NET (p)	Citrix file sharing service API	1653	45	33	471	15
storage (p)	Manage Azure storage services (azure-sdk-for-net)	5487	122	37	270	28
tweetinvi	Library to access Twitter REST API	30 267	99	1283	8485	266
Total		87 773	838	2666	17 422	468

Table 2: Summary of the experimental results.

Benchmark	Time (m)	Reports before filtering	Reports after filtering	Bugs	Dev. fixes
amqpnetlite	15	9	5	3	3
authentication	17	4	1	0	0
autorest.core	9	1	1	1	0
dinero	6	28	22 (3)	3	3
dotnetty	57	0	0	0	0
hbase-sdk-for-net	10	60	27	27	27
numerous-app	3	15	1	1	0
o365rwsclient	2	1	1	1	1
sharefile-NET	6	1	1	1	0
storage	52	27	0	0	0
tweetinvi	38	14	7	6	6
Total	215	160	66 (47)	43	40

confirmed bugs. We reported all the 43 confirmed bugs to the library developers. To date, 40 have been confirmed and fixed by them, as reported in the "Dev. fixes" column. Our bug reports and links to their fixes are available at [17].

Effectiveness of Filtering The filtering techniques, discussed in Section 4.4, were quite effective, pruning 94 reports across the benchmarks. The filters were especially useful in the case of hbase-sdk-for-net and storage. In the former case, the strategy of not reporting cycles on blocking procedures in compiler-generated code proved to be the most effective. For the latter, the strategy of considering threads pertaining only to asynchronous execution of continuations helped rule out many false positives.

After filtering, we observed 4 false positives. The main reason for false positives was that some correspondences between blocking and signaling procedures we inferred were spurious owing to imprecision in the points-to sets of their receivers. For the authentication benchmark, the false positive was due to over-approximating the scheduling behaviour of a framework asynchronous procedure, that gave rise to spurious schedules.

Scalability vs Precision Since our focus was to find real bugs with few false positives, we configured Seal to be precise. The heap contexts used were unbounded, and this was another factor contributing to the low false positive rate. However, this makes the analysis expensive and Seal originally timed out in the case of amqpnetlite, sharefile-NET and storage. For them, we only analyzed those benchmark namespaces our clients referred to. In spite of this, DeadWait found real deadlocks in both amqpnetlite and sharefile-NET.

Deadlocks and Fixes The developers of amqpnetlite and o365rwsclient configured `await` statements so that the signaling procedures we flagged as problematic would run on the thread-pool. Other strategies to fix deadlocks were adopted in the remaining cases. The developers of dinero and tweetinvi undertook a significant rewriting of many procedures. For example, the developer of tweetinvi used the `Task::Run` method to ensure that signaling procedures would not depend on blocked threads. Since almost all the synchronous APIs of hbase-sdk-for-net were susceptible to deadlocks, they have now been removed.

Soundness We inherit the limitations of Seal [53] and do not handle reflection, and calls to native and GUI libraries soundly. Seal does not analyze all framework namespaces fully. For example, it uses stubs for native procedures. Another potential source of unsoundness is that for 3 of the 11 benchmarks (marked by "(p)" in Table 1), we only analyze those benchmark namespaces our clients refer to. Unanalyzed procedures do not affect the input heap in Seal, and they may cause DeadWait to miss deadlocks that depend on specific heap configurations set up by such procedures. Our modeling of framework asynchronous procedures assumes that continuations on these procedures will only resume on the thread-pool. We may miss deadlocks if this assumption is violated, or a client could block all thread-pool threads.

6.2 Future Tool Extensions

It would be interesting to explore other points in the design space, trading precision for scalability, using techniques such as the adaptive k-sensitive may-alias analysis [56]. In our implementation, we have assumed that only `Task` objects can be awaited. However, C$^\sharp$ supports awaiting any object that implements a particular framework defined interface [24]. We plan to relax this assumption in the future, and handle more TPL constructs. The .NET framework for certain domains, such as ASP.NET for web apps, uses sophisticated schedulers. Their use can influence the threads on which continuations run, in ways not captured by our program model. Handling them is promising future work.

7. Related Work

Asynchronous Programming Language support for asynchrony in C++ [39], Dart [26], ECMAScript [13], F$^\sharp$ [61], PHP [16], Python [59] and Scala [40] uses some form of the Future [41, 52] type. Objects of this type are used to represent the future result of asynchronous computations. While C++, C$^\sharp$, PHP and Scala also allow synchronous blocking on the result (e.g., using `Task::Wait` in C$^\sharp$), the other languages only permit registering callbacks with the objects of this type at present. The callbacks accept the future result as a parameter. Typically, registration of distinct callbacks on successful or exceptional computation are supported.

P [32] and P# [30] support state machine modeling to write asynchronous event driven code, and provide runtime support for systematic testing. Other work [22, 36, 37, 44, 60] considers programs where procedures can make explicitly tagged asynchronous calls, that return immediately, and addresses dataflow analysis or verification of such programs. Kahlon et al. [49] propose a static analysis for event-driven C programs with event handlers represented by function pointers, to detect race conditions. Madsen et al. [54] represent single-threaded asynchronous JavaScript applications using event-based call graphs and use them to find bugs in event handling. Similar to CSGs, the event-based call graphs are more expressive than traditional call graphs, tracking, for example, the registration and raising of events. However, we work with implicitly registered callbacks rather than explicitly registered events, and model asynchronous procedures, that are not atomic and can suspend, using state machines. Moreover, our work is targeted at deadlock detection.

Blocking on Futures de Boer et al. [29] identify the potential for deadlock when combining blocking and non-blocking mechanisms to access futures, and model deadlock detection as a reachability problem in Petri nets. Deco [35] addresses the same problem by using alias analysis to construct a dependency graph. Cycles in this graph represent deadlocks, and infeasible cycles are pruned using a may-happen-in-parallel analysis [20]. While the cause of deadlock in our case is similar, Deco targets research languages that combine actors and object oriented features [45, 46]. In the concurrency model considered by [20, 35], a group of objects is associated with a processor, and scheduling tasks onto objects is syntactically explicit in source code. Once scheduled onto an object, a task never migrates, even if it suspends multiple times. Objects cannot directly access fields of other objects, and assignment between task variables is not permitted. These restrictions are absent in asynchronous C$^\sharp$ programs we handle.

Deadlock Detection There is a rich body of work on deadlock detection for multi-threaded programs, using model checkers [28, 31, 38], dynamic analysis [23, 27, 34, 42, 47], static analysis [25, 33, 48, 56, 64], combinations of static and dynamic analysis [18, 19] and runtime monitors [21, 43, 51]. Our work is most closely related to the work of Naik et al. on static deadlock detection [56] and race detection [55] for Java. The former uses a combined points-to and call graph analysis to identify pairs of threads and lock acquisition statements that may deadlock. Our work also relies on call graphs and points-to analysis to identify candidate deadlocks. However, their approach [56] focuses on deadlocks that arise in multi-threaded programs due to circular dependencies in lock acquisition.

Okur et al. [57] design a refactoring tool to convert callback-based asynchronous code to use `async/await`. In addition, they find and correct likely misuses of `async/await`. They do not aim to detect real deadlocks; rather, they attempt to preemptively correct potential deadlocks. The correction they suggest is to use `ConfigureAwait(false)` to force the continuations to run on thread-pool threads, and thus reduce the chances for a deadlock. However, the suggested correction is only applicable in cases where the continuation does not perform thread-unsafe accesses. Their heuristics to detect thread-safety are limited to UI objects, and are based on pattern matching. In contrast, we do whole program analysis to detect deadlocks.

8. Conclusions

Asynchronous programming, through the use of modern `async/await` primitives, is seeing widespread adoption. In this paper, we study the problem of deadlocks in asynchronous C$^\sharp$ programs, and present the first deadlock detection technique for them. Towards this, we model the mixed synchronous and asynchronous control flow using the continuation scheduling graph. We identify deadlocks by analyzing the inter-dependence between blocking and signaling procedures, and the threads they may execute on. We believe that CSGs can serve as a useful representation for other program analyses of asynchronous programs.

We implemented our static analysis in a tool called DeadWait. Using DeadWait, we analyzed 11 asynchronous C$^\sharp$ libraries, finding 43 deadlocks, of which developers fixed 40. This provides a practical validation of existence of deadlocks in asynchronous programs, and the utility of DeadWait in finding them. We have outlined possible extensions to our work in the Future Tool Extensions section.

Acknowledgments We thank Ravichandran Madhavan for many invigorating technical discussions, and his help in adapting Seal. We thank Madan Musuvathi, our shepherd, and the anonymous reviewers for their insightful feedback and advice on improving the paper.

References

[1] Asynchronous programming model (APM). https://msdn.microsoft.com/en-us/library/ms228963(v=vs.110).aspx.

[2] Event-based asynchronous pattern (EAP). https://msdn.microsoft.com/en-us/library/ms228969(v=vs.110).aspx.

[3] Sharefile. https://github.com/citrix/ShareFile-NET.

[4] Tweetinvi. https://github.com/linvi/tweetinvi.

[5] Amqpnetlite. https://github.com/Azure/amqpnetlite.

[6] Autorest. https://github.com/Azure/autorest.

[7] azure-sdk-for-net. https://github.com/Azure/azure-sdk-for-net.

[8] Walkthrough: Accessing the web by using async and await. https://msdn.microsoft.com/en-us/library/hh300224.aspx.

[9] Dinero. https://github.com/DineroRegnskab/dinero-csharp-sdk.

[10] Dotnetty. https://github.com/Azure/dotnetty.

[11] APIs, SDKs and open source projects from Microsoft Azure. https://github.com/azure.

[12] Hbase. https://github.com/hdinsight/hbase-sdk-for-net.

[13] Async functions for ECMAScript. https://github.com/tc39/ecmascript-asyncawait.

[14] Numerousapp. https://github.com/ebezine/numerousapp-net.

[15] O365rwsclient. https://github.com/Microsoft/o365rwsclient.

[16] Async: Introduction. https://docs.hhvm.com/hack/async.

[17] Deadlocks reported and fixed. http://www.iisc-seal.net/deadwait.

[18] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, pages 191–207. Springer-Verlag, 2006.

[19] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM J. Res. Dev.*, 54(5):520–534, Sept. 2010.

[20] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. May-happen-in-parallel analysis for actor-based concurrency. *ACM Trans. Comput. Logic*, 17(2):11:1–11:39, Dec. 2015.

[21] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pages 143–162. ACM, 2008.

[22] M. F. Atig, A. Bouajjani, and T. Touili. Analyzing Asynchronous Programs with Preemption. volume 2 of *FSTTCS*, pages 37–48. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008.

[23] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, pages 208–223. Springer-Verlag, 2006.

[24] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause 'N' Play: Formalizing asynchronous C#. In *ECOOP*, pages 233–257. Springer-Verlag, 2012.

[25] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002.

[26] G. Bracha. Dart language asynchrony support: Phase 1. https://www.dartlang.org/articles/await-async/.

[27] Y. Cai, S. Wu, and W. K. Chan. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *ICSE*, pages 491–502. ACM, 2014.

[28] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[29] F. de Boer, M. Bravetti, I. Grabe, M. Lee, M. Steffen, and G. Zavattaro. A Petri Net based analysis of deadlocks for active objects and futures. In *Formal Aspects of Component Software*, volume 7684 of *Lecture Notes in Computer Science*, pages 110–127. Springer Berlin Heidelberg, 2013.

[30] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous programming, analysis and testing with state machines. In *PLDI*, pages 154–164. ACM, 2015.

[31] C. DeMartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Softw. Pract. Exper.*, 29 (7):577–603, June 1999.

[32] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *PLDI*, pages 321–332. ACM, 2013.

[33] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252. ACM, 2003.

[34] M. Eslamimehr and J. Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In *FSE*, pages 353–365. ACM, 2014.

[35] A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems*, volume 7892 of *Lecture Notes in Computer Science*, pages 273–288. Springer Berlin Heidelberg, 2013.

[36] P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1): 6:1–6:48, May 2012.

[37] I. Gavran, F. Niksic, A. Kanade, R. Majumdar, and V. Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *CONCUR*, pages 483–496, 2015.

[38] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186. ACM, 1997.

[39] N. Gustafsson, D. Brewis, and H. Sutter. Resumable functions. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3858.pdf.

[40] P. Haller and J. Zaugg. Scala Improvement Process - Async. http://docs.scala-lang.org/sips/pending/async.html.

[41] R. H. Halstead, Jr. MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7 (4):501–538, Oct. 1985.

[42] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264. Springer-Verlag, 2000.

[43] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *PLDI*, pages 141–152. ACM, 2013.

[44] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, pages 339–350. ACM, 2007.

[45] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1):23–66, Nov. 2006.

[46] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of the 9th International Conference on Formal Methods for Components and Objects*, pages 142–164. Springer-Verlag, 2011.

[47] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120. ACM, 2009.

[48] V. Kahlon, F. Ivančić, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518. Springer-Verlag, 2005.

[49] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *FSE*, pages 13–22. ACM, 2009.

[50] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, pages 227–242. ACM, September 2009.

[51] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the USENIX Annual Technical Conference*, pages 3–3. USENIX Association, 2005.

[52] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, pages 260–267. ACM, 1988.

[53] R. Madhavan, G. Ramalingam, and K. Vaswani. Modular heap analysis for higher-order programs. In *SAS*, pages 370–387. Springer-Verlag, 2012.

[54] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven node.js JavaScript applications. In *OOPSLA*, pages 505–519. ACM, 2015.

[55] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319. ACM, 2006.

[56] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396. IEEE Computer Society, 2009.

[57] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen. A study and toolkit for asynchronous programming in C#. In *ICSE*, pages 1117–1127. ACM, 2014.

[58] A. Santhiar and A. Kanade. Semantics of Asynchronous C$^\sharp$. http://www.iisc-seal.net/publications/asyncsemantics.pdf.

[59] Y. Selivanov. Pep 492 – coroutines with async and await syntax. https://www.python.org/dev/peps/pep-0492/#id27.

[60] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, pages 300–314. Springer-Verlag, 2006.

[61] D. Syme, T. Petricek, and D. Lomov. The F# Asynchronous Programming Model. In *PADL*, 2011.

[62] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[63] S. Toub. Await, and UI, and deadlocks! Oh my! http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/await-and-ui-and-deadlocks-oh-my.aspx.

[64] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP*, pages 602–629. Springer-Verlag, 2005.