Automated Synthesis of Asynchronizations

Sidi Mohamed Beillahi[1(✉)], Ahmed Bouajjani[2], Constantin Enea[3], and Shuvendu Lahiri[4]

[1] University of Toronto, Toronto, Canada
sm.beillahi@utoronto.ca
[2] Université Paris Cité, IRIF, CNRS, Paris, France
abou@irif.fr
[3] LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, Paris, France
cenea@lix.polytechnique.fr
[4] Microsoft Research Lab - Redmond, Redmond, USA
shuvendu@microsoft.com

Abstract. Asynchronous programming is widely adopted for building responsive and efficient software, and modern languages such as C# provide async/await primitives to simplify the use of asynchrony. In this paper, we propose an approach for refactoring a sequential program into an asynchronous program that uses async/await, called asynchronization. The refactoring process is parametrized by a set of methods to replace with asynchronous versions, and it is constrained to avoid introducing data races. We investigate the delay complexity of enumerating all data race free asynchronizations, which quantifies the delay between outputting two consecutive solutions. We show that this is polynomial time modulo an oracle for solving reachability in sequential programs. We also describe a pragmatic approach based on an interprocedural dataflow analysis with polynomial-time delay complexity. The latter approach has been implemented and evaluated on a number of non-trivial C# programs extracted from open-source repositories.

1 Introduction

Asynchronous programming is widely adopted for building responsive and efficient software. As an alternative to explicitly registering callbacks with asynchronous calls, C# 5.0 [4] introduced the `async/await` primitives. These primitives allow the programmer to write code in a familiar sequential style without explicit callbacks. An asynchronous procedure, marked with `async`, returns a task object that the caller uses to "await" it. Awaiting may suspend the execution of the caller, but does not block the thread it is running on. The code after `await` is the continuation called back when the callee result is ready. This paradigm has become popular across many languages, C++, JavaScript, Python.

The `async/await` primitives introduce concurrency which is notoriously complex. The code in between a call and a matching `await` (referring to the same

G. Singh and C. Urban (Eds.): SAS 2022, LNCS 13790, pp. 135–159, 2022.
https://doi.org/10.1007/978-3-031-22308-2_7

```
 1 void Main(string f) {
 2   x = 0;
 3   int val = RdFile(f);
 4   y = 1;

 6   int r = x;
 7   Debug.Assert(r == val); }

 9 int RdFile(string f) {
10   var rd=new StreamReader(f);
11   string s = rd.ReadToEnd();
12   int r1 = x;

14   x = r1 + s.Length;
15   return s.Length;    }
```

```
 1 async Task Main(string f) {
 2   x = 0;
 3   Task<int> t1 = RdFile(f);
 4   y = 1;
 5   int val = await t1;
 6   int r = x;
 7   Debug.Assert(r == val);  }

 9 async Task<int> RdFile(string f) {
10   var rd = new StreamReader(f);
11   Task<string> t=rd.ReadToEndAsync();
12   int r1 = x;
13   string s = await t;
14   x = r1 + s.Length;
15   return s.Length;              }
```

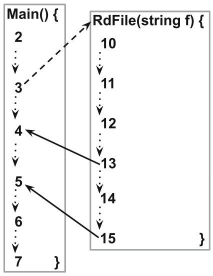

Fig. 1. Synchronous and asynchronous C# programs (`x`, `y` are static variables).

task) may execute before some part of the awaited task or after the awaited task finished. For instance, on the middle of Fig. 1, the assignment `y=1` at line 4 can execute before or after `RdFile` finishes. The `await` for `ReadToEndAsync` in `RdFile` (line 13) may suspend `RdFile`'s execution because `ReadToEndAsync` did not finish, and pass the control to `Main` which executes `y=1`. If `ReadToEndAsync` finishes before this `await` executes, then the latter has no effect and `y=1` gets executed after `RdFile` finishes. The resemblance with sequential code can be especially deceitful since this non-determinism is opaque. It is common that `awaits` are placed immediately after the corresponding call which limits the benefits that can be obtained from executing steps in the caller and callee concurrently [25].

In this paper, we address the problem of writing efficient asynchronous code that uses `async`/`await`. We propose a procedure for automated synthesis of asynchronous programs *equivalent* to a given synchronous (sequential) program P. This can be seen as a way of refactoring synchronous code to asynchronous code. Solving this problem in its full generality would require checking equivalence between arbitrary programs, which is known to be hard. Therefore, we consider a restricted space of asynchronous program candidates defined by substituting synchronous methods in P with asynchronous versions (assumed to be behaviorally equivalent). The substituted methods are assumed to be leaves of the call-tree (they do not call any method in P). Such programs are called *asynchronizations* of P. A practical instantiation is replacing IO synchronous calls for reading/writing files or managing http connections with asynchronous versions.

For instance, the sequential C# program on the left of Fig. 1 contains a `Main` that invokes a method `RdFile` that returns the length of the text in a file. The file name input to `RdFile` is an input to `Main`. The program uses a variable `x` to aggregate the lengths of all files accessed by `RdFile`; this would be more useful when `Main` calls `RdFile` multiple times which we omit for simplicity. Note that this program passes the assertion at line 7. The time consuming method `ReadToEnd` for reading a file is an obvious choice for being replaced with an equivalent *asynchronous* version whose name is suffixed with `Async`. Performing such tasks asynchronously can lead to significant performance boosts. The program on the middle of Fig. 1 is an example of an asynchronization defined by this substitution. The syntax of `async`/`await` imposes that every method that transitively calls one of the substituted methods, i.e., `Main` and `RdFile`, must also be declared as asynchronous. Then, every asynchronous call must be followed by an `await` that specifies the control location where that task should have completed. For instance, the `await` for `ReadToEndAsync` is placed at line 13 since the next instruction (at line 14) uses the computed value. Therefore, synthesizing such refactoring reduces to finding a correct placement of `awaits` (that implies equivalence) for every call of a method that transitively calls a substituted method (we do not consider "deeper" refactoring like rewriting conditionals or loops).

We consider an equivalence relation between a synchronous program and an asynchronization that corresponds to absence of data races in the asynchronization. Data race free asynchronizations are called *sound*. Relying on absence of data races avoids reasoning about equality of sets of reachable states which is harder in general, and an established compromise in reasoning about concurrency. For instance, the asynchronization in Fig. 1 is sound because the call to `RdFile` accessing `x` finishes before the read of `x` in `Main` (line 6). Therefore, accesses to `x` are performed in the same order as in the synchronous program.

The asynchronization on the right of Fig. 1 is not the only sound (data-race free) asynchronization of the program on the left. The `await` at line 13 can be moved one statement up (before the read of `x`) and the resulting program remains equivalent to the sequential one. In this paper, we investigate the problem of enumerating *all* sound asynchronizations of a sequential program P w.r.t. substituting a set of methods with asynchronous versions. This makes it possible to deal separately with the problem of choosing the best asynchronization in terms of performance based on some metric (e.g., performance tests).

Identifying the most efficient asynchronization is difficult and can not be done syntactically. It is tempting to consider that increasing the distance between calls and matching `awaits` so that more of the caller code is executed while waiting for an asynchronous task to finish increases performance. However, this is not true in general. We use the programs in Fig. 2 to show that the best `await` placement w.r.t. performance depends on execution times of code blocks in between calls and `awaits` in a non-trivial manner. Note that estimating these execution times, especially for IO operations like http connections, can not be done statically.

The programs in Fig. 2 use `Thread.Sleep(n)` to abstract sequential code executing in n milliseconds and `Task.Delay(n)` to abstract an asynchronous call executing in n milliseconds on a different thread. The functions named `Foo` differ only in the position of `await t`. We show that modifying this position worsens execution time in each case. For the left program, best performance corresponds to maximal distance between `await t` in `Foo` and the corresponding call. This allows the `IO` call to execute in parallel with the caller, as depicted on the bottom-left of Fig. 2. The executions corresponding to the other two positions of `await t` are given just above. For the middle program, placing `await t` in between the two code blocks in `Foo` optimizes performance (note the extra `IO` call in `Main`): the `IO` call in `Foo` executes in parallel with the first code block in `Foo` and the `IO` call in `Main` executes in parallel with the second one. This is depicted on the bottom-middle of Fig. 2. The execution above shows that placing `await t` as on the left (after the two code blocks) leads to worse execution time (placing `await t` immediately after the call is also worse). Finally, for the right program, placing

```
async Task Main() {
 var t1 = Foo();

 await t1;
                   }

async Task Foo() {
 var t = IO();

 Thread.Sleep(200);

 Thread.Sleep(200);
 await t;         }

async Task IO() {
 var t0 = Task.Delay(300);

 await t0;      }
```

```
async Task Main() {
 var t1 = Foo();
 var t2 = IO();
 await t1;
 await t2;        }

async Task Foo() {
 var t = IO();

 Thread.Sleep(200);
 await t;
 Thread.Sleep(200);
                   }

async Task IO() {
 var t0 = Task.Delay(300);

 await t0;      }
```

```
async Task Main() {
 var t1 = Foo();
 var t2 = IO();
 await t1;
 await t2;        }

async Task Foo() {
 var t = IO();
 await t;
 Thread.Sleep(200);

 Thread.Sleep(200);
                   }

async Task IO() {
 var t0 = Task.Delay(300);
 Thread.Sleep(150);
 await t0;      }
```

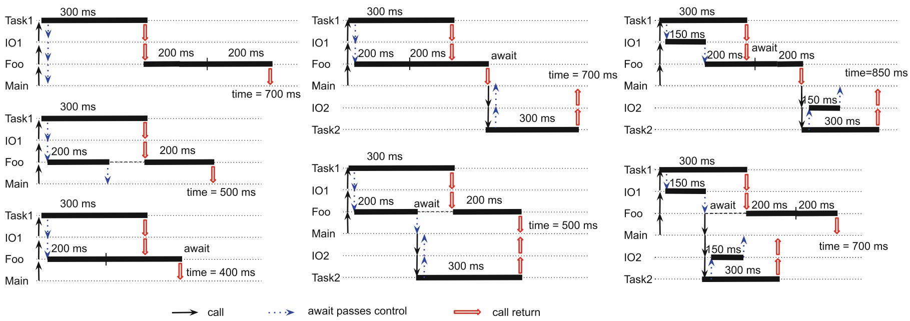

Fig. 2. Asynchronous C# programs and executions. On the bottom, time durations of executing code blocks from the same method are aligned horizontally, and time goes from left to right. Vertical single-line arrows represent method call steps, dashed arrows represent `awaits` passing control to the caller, and double-line arrows represent a call return. Total execution time is marked `time=...`.

`await t` immediately after the call is best (note that `IO` executes another code block before `await`). The `IO` call in `Main` executes in parallel with `Foo` as shown on the bottom-right of Fig. 2. The execution above shows the case where `await t` is placed in the middle (the await has no effect because `IO` already finished, and `Foo` continues to execute). This leads to worse execution time (placing `await t` after the two code blocks is also worse). These differences in execution times have been confirmed by running the programs on a real machine.

As demonstrated by the examples in Fig. 2, the performance of an asynchronization depends on the execution environment, e.g., the overhead of IO operations like http connections and disk access (in Fig. 2, we use `Thread.Sleep(n)` or `Task.Delay(n)` to model such overheads). Since modeling the behavior of an execution environment w.r.t. performance is difficult in general, selecting the most performant asynchronization using static reasoning is also difficult. As a way of sidestepping this difficulty, we focus on enumerating *all* sound asynchronizations that allows to evaluate performance separately in a dynamic manner using performance tests for instance (for each sound asynchronization).

In the worst-case, the number of (sound) asynchronizations is exponential in the number of method calls in the program. Therefore, we focus on the *delay complexity* of the problem of enumerating sound asynchronizations, i.e., the complexity of the delay between outputting two consecutive (distinct) solutions, and show that this is polynomial time modulo an oracle for solving reachability (assertion checking) in *sequential* programs. Note that a trivial enumeration of all asynchronizations and checking equivalence for each one of them has an exponential delay complexity modulo an oracle for checking equivalence.

As an intermediate step, we consider the problem of computing *maximal* sound asynchronizations that maximize the distance between every call and its matching `await`. We show that rather surprisingly, there exists a *unique* maximal sound asynchronization. This is not trivial since asynchronizations can be incomparable w.r.t. distances between calls and `await`s (i.e., better for one `await` and worse for another, and vice-versa). This holds even if maximality is relative to a given asynchronization P_a imposing an upper bound on the distance between awaits and calls. In principle, avoiding data races could reduce to a choice between moving one await or another closer to the matching call. We show that this is not necessary because the maximal asynchronization is required to be equivalent to a *sequential* program, which executes statements in a fixed order.

As a more pragmatic approach, we define a procedure for computing sound asynchronizations which relies on a bottom-up interprocedural data-flow analysis. The placement of awaits is computed by traversing the call graph bottom up and using a data-flow analysis that computes read or write accesses made in the callees. We show that this procedure computes maximal sound asynchronizations of abstracted programs where every Boolean condition is replaced with a non-deterministic choice. These asynchronizations are sound for the concrete programs as well. This procedure enables a polynomial-time delay enumeration of sound asynchronizations of abstracted programs.

We implemented the asynchronization enumeration based on data-flow analysis in a prototype tool for C# programs. We evaluated this implementation on a number of non-trivial programs extracted from open source repositories to show that our techniques have the potential to become the basis of refactoring tools that allow programmers to improve their usage of async/await primitives.

In summary, this paper makes the following contributions:

– Define the problem of data race-free (sound) asynchronization synthesis for refactoring sequential code to equivalent asynchronous code (Sect. 3).
– Show that the problem of computing a sound asynchronization that maximizes the distance between calls and awaits has a unique solution (Sect. 4).
– The delay complexity of sound asynchronization synthesis (Sects. 5–6).
– A pragmatic algorithm for computing sound asynchronizations based on a data-flow analysis (Sect. 7).
– A prototype implementation of this algorithm and an evaluation of this prototype on a benchmark of non-trivial C# programs (Sect. 8).

Additional formalization and proofs are included in [3].

$$
\begin{array}{lcl}
\langle prog\rangle & ::= & \texttt{program}\ \langle md\rangle \\
\langle md\rangle & ::= & \texttt{method}\ \langle m\rangle\ \{\ \langle inst\rangle\ \}\ |\ \texttt{async method}\ \langle m\rangle\ \{\ \langle inst\rangle\ \}\ |\ \langle md\rangle\ \langle md\rangle \\
\langle inst\rangle & ::= & \langle x\rangle\ \texttt{:=}\ \langle le\rangle\ |\ \langle r\rangle\ \texttt{:=}\ \langle x\rangle\ |\ \langle r\rangle := \texttt{call}\ \langle m\rangle\ |\ \texttt{return}\ |\ \texttt{await}\ \langle r\rangle \\
 & & |\ \texttt{await}\ *\ |\ \texttt{if}\ \langle le\rangle\ \{\langle inst\rangle\}\ \texttt{else}\ \{\langle inst\rangle\}\ |\ \texttt{while}\ \langle le\rangle\ \{\langle inst\rangle\}\ | \\
 & & \langle inst\rangle\ ;\ \langle inst\rangle
\end{array}
$$

Fig. 3. Syntax. $\langle m\rangle$, $\langle x\rangle$, and $\langle r\rangle$ represent method names, program and local variables, resp. $\langle le\rangle$ is an expression over local variables, or $*$ which is non-deterministic choice.

2 Asynchronous Programs

We consider a simple programming language to formalize our approach, shown in Fig. 3. A *program* is a set of methods, including a distinguished `main`, which are classified as *synchronous* or *asynchronous*. Synchronous methods run continuously until completion when they are invoked. Asynchronous methods, marked using the keyword `async`, can run only partially and be interrupted when executing an `await`. Only asynchronous methods can use `await`, and all methods using `await` must be defined as asynchronous. We assume that methods are not (mutually) recursive. A program is called *synchronous* if it is a set of synchronous methods.

A method is defined by a name from a set $\mathbb{M}$ and a list of statements over a set $\mathbb{PV}$ of *program variables*, which can be accessed from different methods (ranged over using x, y, z,...), and a set $\mathbb{LV}$ of method *local variables* (ranged over using r, r_1, r_2,...). Input/return parameters are modeled using program variables. Each method call returns a *unique task identifier* from a set $\mathbb{T}$, used to record control dependencies imposed by `await`s (for uniformity, synchronous methods return a task identifier as well). Our language includes assignments, `await`s, `return`s, loops, and conditionals. Assignments to a local variable $r := x$, where x is a program variable, are called *reads* of x, and assignments to a program variable $x := le$ (le is an expression over local variables) are called *writes* to x. A *base* method is a method whose body does *not* contain method calls.

Asynchronous Methods. Asynchronous methods can use `await`s to wait for the completion of a task (invocation) while *the control is passed to their caller*. The parameter r of the `await` specifies the id of the awaited task. As a sound abstraction of awaiting the completion of an IO operation (reading or writing a file, an http request, etc.), which we do not model explicitly, we use a variation `await` $*$. This has a non-deterministic effect of either continuing to the next statement in the same method (as if the IO operation already completed), or passing the control to the caller (as if the IO operation is still pending).

```
async method ReadToEndAsync() {
 await *;
 ind = Stream.index;
 len = Stream.content.Length;
 if (ind >= len)
  retVal  =  ""; return
 Stream.index = len;
 retVal = Stream.content(ind,len);
 return                        }
```

Fig. 4. An IO method.

Figure 4 lists our modeling of the IO method `ReadToEndAsync` used in Fig. 1. We use program variables to represent system resources such as the file system. The await for the completion of accesses to such resources is modeled by `await`

$*$. This enables capturing racing accesses to system resources in asynchronous executions. Parameters or return values are modeled using program variables. `ReadToEndAsync` is modeled using reads/writes of the index/content of the input stream, and `await` $*$ models the await for their completion.

We assume that the body of every asynchronous method m satisfies several well-formedness syntactic constraints, defined on its control-flow graph (CFG). We recall that each node of the CFG represents a basic block of code (a maximal-length sequence of branch-free code), and nodes are connected by directed edges which represent a possible transfer of control between blocks. Thus,

1. every call $r :=$ `call` m' uses a distinct variable r (to store task identifiers),
2. every CFG block containing an `await` r is dominated by the CFG block containing the call $r :=$ `call` $\ldots$ (i.e., every CFG path from the entry to the await has to pass through the call),
3. every CFG path starting from a block containing a call $r :=$ `call` $\ldots$ to the exit has to pass through an `await` r statement.

The first condition simplifies the technical exposition, while the last two ensure that r stores a valid task identifier when executing an `await` r, and that every asynchronous invocation is awaited before the caller finishes. Languages like C# or Javascript do not enforce the latter constraint, but it is considered bad practice due to possible exceptions that may arise in the invoked task and are not caught. We forbid passing task identifiers as method parameters (which is possible in C#). A statement `await` r is said to *match* a statement $r :=$ `call` m'.

In Fig. 5, we give three examples of programs to explain in more details the well-formedness syntactic constraints. The program on the left of Fig. 5 does not satisfy the second condition since `await r` can be reached without entering the loop. The program in the center of Fig. 5 does not satisfy the third condition since we can reach the end of the method without entering the if branch and thus, without executing `await r`. The program on the right of Fig. 5 satisfies both conditions.

```
async method m {
  while *
    r = call m1;

  await r;
}
```

```
async method m {
  r = call m1;

  if *
    await r;
}
```

```
async method m {
  r = call m1;
  while *
    r' = call m1;
    await r';
  await r;
}
```

Fig. 5. Examples of programs

Semantics. A program configuration is a tuple $(\mathsf{g}, \mathsf{stack}, \mathsf{pend}, \mathsf{cmpl}, \mathsf{c\text{-}by}, \mathsf{w\text{-}for})$ where g is composed of the valuation of the program variables excluding the program counter, stack is the call stack, pend is the set of asynchronous tasks, e.g., continuations predicated on the completion of some method call, cmpl is the set of completed tasks, $\mathsf{c\text{-}by}$ represents the relation between a method call and its caller, and $\mathsf{w\text{-}for}$ represents the control dependencies imposed by `await` statements. The activation frames in the call stack and the asynchronous tasks are represented using triples (i, m, ℓ) where $i \in \mathbb{T}$ is a task identifier, $m \in \mathbb{M}$ is a method name, and ℓ is a valuation of local variables, including as usual a dedicated program counter. The set of completed tasks is represented as a function $\mathsf{cmpl} : \mathbb{T} \to \{\top, \bot\}$ such that $\mathsf{cmpl}(i) = \top$ when i is completed and $\mathsf{cmpl}(i) = \bot$, otherwise. We define $\mathsf{c\text{-}by}$ and $\mathsf{w\text{-}for}$ as partial functions $\mathbb{T} \rightharpoonup \mathbb{T}$

with the meaning that $\mathsf{c\text{-}by}(i) = j$, resp., $\mathsf{w\text{-}for}(i) = j$, iff i is called by j, resp., i is waiting for j. We set $\mathsf{w\text{-}for}(i) = *$ if the task i was interrupted because of an `await` $*$ statement.

The semantics of a program P is defined as a labeled transition system (LTS) $[P] = (\mathbb{C}, \mathbb{A}\mathsf{ct}, \mathsf{ps}_0, \rightarrow)$ where $\mathbb{C}$ is the set of program configurations, $\mathbb{A}\mathsf{ct}$ is a set of transition labels called *actions*, ps_0 is the initial configuration, and $\rightarrow \subseteq \mathbb{C} \times \mathbb{A}\mathsf{ct} \times \mathbb{C}$ is the transition relation. Each program statement is interpreted as a transition in $[P]$. The set of actions is defined by ($\mathbb{A}\mathsf{id}$ is a set of action identifiers):

$$\begin{aligned}\mathbb{A}\mathsf{ct} =& \{(aid, i, ev) : aid \in \mathbb{A}\mathsf{id}, i \in \mathbb{T}, ev \in \{\mathsf{rd}(x), \mathsf{wr}(x), \mathsf{call}(j), \mathsf{await}(k), \mathsf{return},\\ &\mathsf{cont} : j \in \mathbb{T}, k \in \mathbb{T} \cup \{*\}, x \in \mathbb{PV}\}\}\end{aligned}$$

The transition relation $\rightarrow$ is defined in Fig. 6. Transition labels are written on top of $\rightarrow$.

Transitions labeled by $(aid, i, \mathsf{rd}(x))$ and $(aid, i, \mathsf{wr}(x))$ represent a read and a write accesses to the program variable x, respectively, executed by the task (method call) with identifier i. A transition labeled by $(aid, i, \mathsf{call}(j))$ corresponds to the fact that task i executes a method call that results in creating a task j. Task j is added on the top of the stack of currently executing tasks, declared pending (setting $\mathsf{cmpl}(j)$ to $\bot$), and $\mathsf{c\text{-}by}$ is updated to track its caller ($\mathsf{c\text{-}by}(j) = i$). A transition $(aid, i, \mathsf{return})$ represents the return from task i. Task i is removed from the stack of currently executing tasks, and $\mathsf{cmpl}(i)$ is set to $\top$ to record the fact that task i is finished.

A transition $(aid, i, \mathsf{await}(j))$ relates to task i waiting asynchronously for task j. Its effect depends on whether task j is already completed. If this is the case (i.e., $\mathsf{cmpl}[j] = \top$), task i continues and executes the next statement. Otherwise, task i executing the await is removed from the stack and added to the set of pending tasks, and $\mathsf{w\text{-}for}$ is updated to track the waiting-for relationship ($\mathsf{w\text{-}for}(i) = j$). Similarly, a transition $(aid, i, \mathsf{await}(*))$ corresponds to task i waiting asynchronously for the completion of an unspecified task. Non-deterministically, task i continues to the next statement, or task i is interrupted and transferred to the set of pending tasks ($\mathsf{w\text{-}for}(i)$ is set to $*$).

A transition (aid, i, cont) represents the scheduling of the continuation of task i. There are two cases depending on whether i waited for the completion of another task j modeled explicitly in the language (i.e., $\mathsf{w\text{-}for}(i) = j$), or an unspecified task (i.e., $\mathsf{w\text{-}for}(i) = *$). In the first case, the transition is enabled only when the call stack is empty and j is completed. In the second case, the transition is always enabled. The latter models the fact that methods implementing IO operations (waiting for unspecified tasks in our language) are executed in background threads and can interleave with the main thread (that executes the `Main` method). Although this may seem restricted because we do not allow arbitrary interleavings between IO methods and `Main`, this is actually sound when focusing on the existence of data races as in our approach. As shown later in Table 1, any two instructions that follow an `await` $*$ are not happens-before related and form a race.

$$\frac{\mathtt{r := x} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \ell' = \ell[r \mapsto \mathsf{g}(x), \mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \_, \_, \_, \_) \xrightarrow{(aid,\, i,\, \mathsf{rd}(x))} (\mathsf{g}, (i, m, \ell') \circ \mathsf{stack}, \_, \_, \_, \_)}$$

$$\frac{\mathtt{x := le} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))] \quad \mathsf{g}' = \mathsf{g}[x \mapsto \ell(\mathtt{le})]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \_, \_, \_, \_) \xrightarrow{(aid,\, i,\, \mathsf{wr}(x))} (\mathsf{g}', (i, m, \ell') \circ \mathsf{stack}, \_, \_, \_, \_)}$$

$$\frac{\begin{array}{c} r := \mathsf{call}\ m \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \ell_0 = \mathsf{init}(\mathsf{g}, m) \quad j \in \mathbb{T} \text{ fresh} \\ \ell' = \ell[r \mapsto j, \mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))] \quad \mathsf{cmpl}' = \mathsf{cmpl}[j \mapsto \perp] \quad \mathsf{c\text{-}by}' = \mathsf{c\text{-}by}[j \mapsto i] \end{array}}{(\mathsf{g}, (i, m', \ell) \circ \mathsf{stack}, \_, \mathsf{cmpl}, \mathsf{c\text{-}by}, \_) \xrightarrow{(aid,\, i,\, \mathsf{call}(j))} (\mathsf{g}, (j, m, \ell_0) \circ (i, m', \ell') \circ \mathsf{stack}, \_, \mathsf{cmpl}', \mathsf{c\text{-}by}', \_)}$$

$$\frac{\mathsf{return} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \mathsf{cmpl}' = \mathsf{cmpl}[i \mapsto \top]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \_, \mathsf{cmpl}, \_, \_) \xrightarrow{(aid,\, i,\, \mathsf{return})} (\mathsf{g}, \mathsf{stack}, \_, \mathsf{cmpl}', \_, \_)}$$

$$\frac{\mathsf{await}\ \mathtt{r} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \mathsf{cmpl}(\ell(r)) = \top \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \_, \mathsf{cmpl}, \_, \_) \xrightarrow{(aid,\, i,\, \mathsf{await}(\ell(r)))} (\mathsf{g}, (i, m, \ell') \circ \mathsf{stack}, \_, \mathsf{cmpl}, \_, \_)}$$

$$\frac{\begin{array}{c} \mathsf{await}\ \mathtt{r} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \mathsf{cmpl}(\ell(r)) = \perp \quad \mathsf{w\text{-}for}' = \mathsf{w\text{-}for}[i \mapsto \ell(r)] \\ \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))] \end{array}}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pend}, \mathsf{cmpl}, \_, \mathsf{w\text{-}for}) \xrightarrow{(aid,\, i,\, \mathsf{await}(\ell(r)))} (\mathsf{g}, \mathsf{stack}, \{(i, m, \ell')\} \uplus \mathsf{pend}, \mathsf{cmpl}, \_, \mathsf{w\text{-}for}')}$$

$$\frac{\mathsf{await} * \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \_, \_, \_, \_) \xrightarrow{(aid,\, i,\, \mathsf{await}(*))} (\mathsf{g}, (i, m, \ell') \circ \mathsf{stack}, \_, \_, \_, \_)}$$

$$\frac{\mathsf{await} * \in \mathsf{inst}(\ell(\mathsf{pc})) \quad aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \mathsf{w\text{-}for}' = \mathsf{w\text{-}for}[i \mapsto *] \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pend}, \_, \_, \mathsf{w\text{-}for}) \xrightarrow{(aid,\, i,\, \mathsf{await}(*))} (\mathsf{g}, \mathsf{stack}, \{(i, m, \ell')\} \uplus \mathsf{pend}, \_, \_, \mathsf{w\text{-}for}')}$$

$$\frac{aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \mathsf{w\text{-}for}(i) = j \quad \mathsf{cmpl}(j) = \top}{(\mathsf{g}, \epsilon, \{(i, m, \ell)\} \uplus \mathsf{pend}, \mathsf{cmpl}, \_, \mathsf{w\text{-}for}) \xrightarrow{(aid,\, i,\, \mathsf{cont})} (\mathsf{g}, (i, m, \ell), \mathsf{pend}, \mathsf{cmpl}, \_, \mathsf{w\text{-}for})}$$

$$\frac{aid \in \mathbb{A}\mathsf{id} \text{ fresh} \quad \mathsf{w\text{-}for}(i) = *}{(\mathsf{g}, \mathsf{stack}, \{(i, m, \ell)\} \uplus \mathsf{pend}, \_, \_, \mathsf{w\text{-}for}) \xrightarrow{(aid,\, i,\, \mathsf{cont})} (\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pend}, \_, \_, \mathsf{w\text{-}for})}$$

Fig. 6. Program semantics. For a function f, we use $f[a \mapsto b]$ to denote a function g such that $g(c) = f(c)$ for all $c \neq a$ and $g(a) = b$. The function inst returns the instruction at some given control location while next gives the next instruction to execute. We use $\circ$ to denote sequence concatenation and init to denote the initial state of a method call.

By the definition of $\rightarrow$, every action $a \in \mathbb{A}\mathsf{ct} \setminus \{(\_, \_, \mathsf{cont})\}$ corresponds to executing some statement in the program, which is denoted by $\mathsf{S}(a)$.

An execution of P is a sequence $\rho = \mathsf{ps}_0 \xrightarrow{a_1} \mathsf{ps}_1 \xrightarrow{a_2} \dots$ of transitions starting in the initial configuration ps_0 and leading to a configuration ps where the call stack and the set of pending tasks are empty. $\mathbb{C}[P]$ denotes the set of all program variable valuations included in configurations that are reached in executions of P. Since we are only interested in reasoning about the sequence of actions $a_1 \cdot a_2 \cdot \dots$ labeling the transitions of an execution, we will call the latter an execution as well. The set of executions of a program P is denoted by $\mathbb{E}\mathsf{x}(P)$.

Traces. The *trace* of execution $\rho \in \mathbb{E}\mathsf{x}(P)$ is a tuple $\mathsf{tr}(\rho) = (\rho, \mathsf{MO}, \mathsf{CO}, \mathsf{SO}, \mathsf{HB})$ of strict partial orders between the actions in ρ defined in Table 1. The *method invocation order* MO records the order between actions in the same invocation, and the *call order* CO is an extension of MO that additionally orders actions before an invocation with respect to those inside that invocation. The *syn-*

chronous happens-before order SO orders the actions in an execution as if all the invocations were synchronous (even if the execution may contain asynchronous ones). It is an extension of CO where additionally, every action inside a callee is ordered before the actions following its invocation in the caller. The (asynchronous) *happens-before order* HB contains typical control-flow constraints: it is an extension of CO where every action a inside an asynchronous invocation is ordered before the corresponding `await` in the caller, and before the actions following its invocation in the caller if a precedes the first[1] `await` in MO (an invocation can be interrupted only when executing an `await`) or if the callee does not contain an `await` (it is synchronous). $\mathbb{T}\mathsf{r}(P)$ is the set of traces of P.

Table 1. Strict partial orders included in a trace. CO, SO, and HB are the smallest satisfying relations.

$a_1 <_\rho a_2$	a_1 occurs before a_2 in ρ and $a_1 \neq a_2$
$a_1 \sim a_2$	$a_1 = (\_, i, \_)$ and $a_2 = (\_, i, \_)$
$(a_1, a_2) \in \mathsf{MO}$	$a_1 \sim a_2 \wedge a_1 <_\rho a_2$
$(a_1, a_2) \in \mathsf{CO}$	$(a_1, a_2) \in \mathsf{MO} \vee (a_1 = (\_, i, \mathsf{call}(j)) \wedge a_2 = (\_, j, \_))$ $\vee\ (\exists\ a_3.\ (a_1, a_3) \in \mathsf{CO} \wedge (a_3, a_2) \in \mathsf{CO})$
$(a_1, a_2) \in \mathsf{SO}$	$(a_1, a_2) \in \mathsf{CO} \vee (\exists\ a_3.\ (a_1, a_3) \in \mathsf{SO} \wedge (a_3, a_2) \in \mathsf{SO})$ $\vee\ (a_1 = (\_, j, \_) \wedge a_2 = (\_, i, \_) \wedge \exists\ a_3 = (\_, i, \mathsf{call}(j)).\ a_3 <_\rho a_2)$
$(a_1, a_2) \in \mathsf{HB}$	$(a_1, a_2) \in \mathsf{CO} \vee (\exists\ a_3.\ (a_1, a_3) \in \mathsf{HB} \wedge (a_3, a_2) \in \mathsf{HB})$ $\vee\ (\ a_1 = (\_, j, \_) \wedge a_2 = (\_, i, \_) \wedge \exists\ a_3 = (\_, i, \mathsf{await}(j)).\ a_3 <_\rho a_2\ )$ $\vee\ (\ a_1 = (\_, j, \mathsf{await}(i')) \text{ is the first await in } j\ \wedge$ $a_2 = (\_, i, \_) \wedge \exists\ a_3 = (\_, i, \mathsf{call}(j)).\ a_3 <_\rho a_2\ )$ $\vee\ (\ a_1 = (\_, j, \_)\ \wedge\ \nexists\ (\_, j, \mathsf{await}(\_)) \in \rho\ \wedge$ $a_2 = (\_, i, \_) \wedge \exists\ a_3 = (\_, i, \mathsf{call}(j)).\ a_3 <_\rho a_2\ )$

On the right of Fig. 1, we show a trace where two statements (represented by the corresponding lines numbers) are linked by a dotted arrow if the corresponding actions are related by MO, a dashed arrow if the corresponding actions are related by CO but not by MO, and a solid arrow if the corresponding actions are related by the HB but not by CO.

3 Synthesizing Asynchronous Programs

Given a synchronous program P and a subset of *base* methods $L \subseteq P$, our goal is to synthesize *all* asynchronous programs P_a that are equivalent to P and that are obtained by substituting every method in L with an equivalent *asynchronous* version. The base methods are considered to be models of standard library calls (e.g., IO operations) and asynchronous versions are defined by inserting `await` $*$ statements in their body. We use $P[L]$ to emphasize a subset of base methods L in a program P. Also, we call L a *library*. A library is called (a)synchronous when all methods are (a)synchronous.

[1] Code in between two awaits can execute before or after the control is returned to the caller, depending on whether the first awaited task finished or not.

Asynchronizations of a Synchronous Program. Let $P[L]$ be a synchronous program, and L_a a set of asynchronous methods obtained from those in L by inserting at least one `await` $*$ statement in their body (and adding the keyword `sync`). Each method in L_a corresponds to a method in L with the same name, and vice-versa. $P_a[L_a]$ is called an *asynchronization* of $P[L]$ with respect to L_a if it is a syntactically correct program obtained by replacing the methods in L with those in L_a and adding `await` statements as necessary.

More precisely, let $L^* \subseteq P$ be the set of all methods of P that transitively call methods of L. Formally, L^* is the smallest set of methods that includes L and satisfies the following: if a method m calls $m' \in L^*$, then $m \in L^*$. Then, $P_a[L_a]$ is an *asynchronization* of $P[L]$ w.r.t. L_a if it is obtained from P as follows:

```
method m {            async method m {       async method m {
  r1 = call m1;         r1 = call m1;          r1 = call m1;
                        await r1;
  r2 = x;               r2 = x;                r2 = x;
            }                           }      await r1;       }
method m1 {           async method m1 {      async method m1 {
                        await *                await *
  retVal = x;           retVal = x;            retVal = x;
  x = input;            x = input;             x = input;
  return;  }            return;      }         return;      }
```

Fig. 7. A program and its asynchronizations.

- Each method in L is replaced with the corresponding method from L_a.
- All methods in $L^* \setminus L$ are declared as asynchronous (because every call to an asynchronous method is followed by an `await` and any method using `await` must be asynchronous).
- For each invocation $r :=$ call m of $m \in L^*$, add `await` statements `await` r satisfying the well-formedness syntactic constraints described in Sect. 2.

Figure 7 lists a synchronous program and its two asynchronizations, where $L = \{m1\}$ and $L^* = \{m, m1\}$. Asynchronizations differ only in the await placement.

$\mathsf{Asy}[P, L, L_a]$ is the set of all asynchronizations of $P[L]$ w.r.t. L_a. The *strong* asynchronization $\mathsf{strongAsy}[P, L, L_a]$ is an asynchronization where every `await` *immediately* follows the matching call. It reaches exactly the same set of program variable valuations as P.

Problem Definition. We investigate the problem of enumerating *all* asynchronizations of a given program w.r.t. a given asynchronous library, which are *sound*, in the sense that they do not admit data races. Two actions a_1 and a_2 in a trace $\tau = (\rho, \mathsf{MO}, \mathsf{CO}, \mathsf{SO}, \mathsf{HB})$ are *concurrent* if $(a_1, a_2) \notin \mathsf{HB}$ and $(a_2, a_1) \notin \mathsf{HB}$.

An ansynchronous program P_a *admits a data race* (a_1, a_2), where $(a_1, a_2) \in \mathsf{SO}$, if a_1 and a_2 are two concurrent actions of a trace $\tau \in \mathbb{T}\mathrm{r}(P_a)$, and a_1 and a_2 are read or write accesses to the same program variable x, and at least one of them is a write. We write data races as ordered pairs w.r.t. SO to simplify the definition of the algorithms in the next sections. Also, note that traces of *synchronous* programs can *not* contain concurrent actions, and therefore they do not admit data races. $\mathsf{strongAsy}[P, L, L_a]$ does not admit data races as well.

$P_a[L_a]$ is called *sound* when it does not admit data races. The absence of data races implies equivalence to the original program, in the sense of reaching the same set of configurations (program variable valuations).

Definition 1. *For a synchronous program $P[L]$ and asynchronous library L_a, the* asychronization synthesis problem *asks to enumerate all sound asynchronizations in* $\mathsf{Asy}[P, L, L_a]$.

4 Enumerating Sound Asynchronizations

We present an algorithm for solving asynchronization synthesis, which relies on a partial order between asynchronizations that guides the enumeration of possible solutions. The partial order takes into account the distance between calls and corresponding awaits. Figure 8 pictures the partial order for asynchronizations of the program on the left of Fig. 1. Each asynchronization is written as a vector of distances, the first (second) element is the number of statements between `await t1` (`await t`) and the matching call (we count only statements that appear in the sequential program). The edges connect comparable elements, smaller elements being below bigger elements. The asynchronization on the middle of Fig. 1 corresponds to the vector $(1, 1)$. The highlighted elements constitute the set of all sound asynchronizations. The strong asynchronization corresponds to the vector $(0, 0)$.

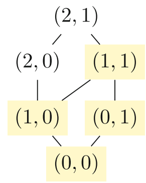

Fig. 8. The partially-ordered set of asynchronizations of the program on the left of Fig. 1.

Formally, an `await` statement s_w in a method m of an asynchronization $P_a[L_a] \in \mathsf{Asy}[P, L, L_a]$ *covers* a read/write statement s in P if there exists a path in the CFG of m from the call statement matching s_w to s_w that contains s. The set of statements covered by an await s_w is denoted by $\mathsf{Cover}(s_w)$. We compare asynchronizations in terms of sets of statements covered by awaits that match the same call from the synchronous program $P[L]$. Since asynchronizations are obtained by adding `await`s, every call in asynchronization $P_a[L_a] \in \mathsf{Asy}[P, L, L_a]$ corresponds to a *fixed* call in $P[L]$. Therefore, for two asynchronizations $P_a, P'_a \in \mathsf{Asy}[P, L, L_a]$, P_a is *smaller* than P'_a, denoted by $P_a \leq P'_a$, iff for every `await` s_w in P_a, there exists an `await` s'_w in P'_a that matches the same call as s_w, such that $\mathsf{Cover}(s_w) \subseteq \mathsf{Cover}(s'_w)$. For example, the two asynchronous programs in Fig. 7 are ordered by $\leq$ since $\mathsf{Cover}(\mathsf{await}\ r1) = \{\}$ in the first and $\mathsf{Cover}(\mathsf{await}\ r1) = \{\texttt{r2 = x}\}$ in the second. Note that the strong asynchronization is smaller than every other asynchronization. Also, note that $\leq$ has a unique maximal element that is called the weakest asynchronization and denoted by $\mathsf{wkAsy}[P, L, L_a]$. In Fig. 8, the weakest asynchronization corresponds to the vector $(2, 1)$.

In the following, we say *moving an await down (resp., up)* when moving the await further away from (resp. closer to) the matching call while preserving well-formedness conditions in Sect. 2. Further away or closer to means increasing or decreasing the set of statements that are covered by the await. For instance, if an await s_w in a program P_a is preceded by a while loop, then *moving it up* means moving it before the whole loop and not inside the loop body. Otherwise, the third well-formedness condition would be violated.

Relative Maximality. A crucial property of this partial order is that for every asynchronization P_a, there exists a *unique* maximal asynchronization that is smaller than P_a and that is sound. Formally, an asynchronization P'_a is called a *maximal asynchronization of P relative to P_a* if (1) $P'_a \leq P_a$, P'_a is sound, and (2) $\forall\ P''_a \in \mathsf{Asy}[P, L, L_a]$. P''_a is sound and $P''_a \leq P_a \Rightarrow P''_a \leq P'_a$.

Algorithm 1. An algorithm for enumerating all sound asynchronizations (these asynchronizations are obtained as a result of the **output** instruction). MAXREL returns the maximal asynchronization of P relative to P_a

procedure ASYSYN(P_a, s_w)
  $P'_a \leftarrow$ MAXREL(P_a);
  output P'_a;
  $\mathcal{P} \leftarrow \mathsf{ImPred}(P'_a, s_w)$;
  for each $(P''_a, s''_w) \in \mathcal{P}$
    ASYSYN(P''_a, s''_w);

Lemma 1. *Given an asynchronization* $P_a \in \mathsf{Asy}[P, L, L_a]$*, there exists a unique program* P'_a *that is a maximal asynchronization of* P *relative to* P_a*.*

The asynchronization P'_a exists because the bottom element of $\leq$ is sound. To prove uniqueness, assume by contradiction that there exist two incomparable maximal asynchronizations P^1_a and P^2_a and select the first await s^1_w w.r.t. the control-flow of the sequential program that is placed in different positions in the two programs. Assume that s^1_w is closer to its matching call in P^1_a. Then, we move s^1_w in P^1_a further away from its matching call to the same position as in P^2_a. This modification does not introduce data races since P^2_a is data race free. Thus, the resulting program is data race free, bigger than P^1_a, and smaller than P_a w.r.t. $\leq$ contradicting the fact that P^1_a is a maximal asynchronization.

4.1 Enumeration Algorithm

Our algorithm for enumerating all sound asynchronizations is given in Algorithm 1 as a recursive procedure ASYSYN that we describe in two phases.

First, ignore the second argument of ASYSYN (in blue), which represents an `await` statement. For an asynchronization P_a, ASYSYN outputs *all* sound asynchronizations that are smaller than P_a. It uses MAXREL to compute the maximal asynchronization P'_a of P relative to P_a, and then, calls itself recursively for all immediate predecessors of P'_a. ASYSYN outputs all sound asynchronizations of P when given as input the weakest asynchronization of P.

Recursive calls on immediate predecessors are necessary because the set of sound asynchronizations is not downward-closed w.r.t. $\leq$. For instance, the asynchronization on the right of Fig. 9 is an immediate predecessor of the sound asynchronization on the left but it has a data race on x.

```
async method m {
  r1 = call m1;
  r2 = x;
  await r1;      }
async method m1 {
  r3 = call m2;
  x = x + 1;
  await r3;      }
async method m2 {
  await *
  retVal = input;
  return;        }
```

```
async method m {
  r1 = call m1;
  r2 = x;
  await r1;      }
async method m1 {
  r3 = call m2;
  await r3;
  x = x + 1;     }
async method m2 {
  await *
  retVal = input;
  return;        }
```

Fig. 9. Asynchronizations.

The delay complexity of this algorithm remains exponential in general, since a sound asynchronization may be outputted multiple times. Asynchronizations are only partially ordered by $\leq$ and different chains of recursive calls starting in different immediate predecessors may end up outputting the same solution. For instance, for the asynchronizations in Fig. 8, the asynchronization $(0, 0)$ will be outputted twice because it is an immediate predecessor of both $(1, 0)$ and $(0, 1)$.

To avoid this redundancy, we use a refinement of the above that *restricts* the set of immediate predecessors available for a (recursive) call of AsySyn. This is based on a *strict total order* $\prec_w$ between `awaits` in a program P_a that follows a topological ordering of its inter-procedural CFG, i.e., if s_w occurs before s'_w in the body of a method m, then $s_w \prec_w s'_w$, and if s_w occurs in a method m and s'_w occurs in a method m' s.t. m (indirectly) calls m', then $s_w \prec_w s'_w$. Therefore, AsySyn takes an await statement s_w as a second parameter, which is initially the maximal element w.r.t. $\prec_w$, and it calls itself only on immediate predecessors of a solution obtained by *moving up* an await s''_w *smaller than or equal to* s_w w.r.t. $\prec_w$. The recursive call on that predecessor will receive as input s''_w. Formally, this relies on a function ImPred that returns pairs of immediate predecessors and await statements defined as follows:

$$\mathsf{ImPred}(P'_a, s_w) = \{(P''_a, s''_w) : P''_a < P'_a \text{ and } \forall P'''_a \in \mathsf{Asy}[P, L, L_a].P'''_a < P'_a \Rightarrow P'''_a \leq P''_a \\ \text{and } s''_w \preceq_w s_w \text{ and } P''_a \in P'_a \uparrow s''_w \}$$

($P'_a \uparrow s''_w$ is the set of asynchronizations obtained from P'_a by changing *only* the position of s''_w, moving it up w.r.t. the position in P'_a). For instance, looking at immediate predecessors of $(1, 1)$ in Fig. 8, $(0, 1)$ is obtained by moving the *first* await in $\prec_w$. Therefore, the recursive call on $(0, 1)$ computes the maximal asynchronization relative to $(0, 1)$, which is $(0, 1)$, and stops (ImPred returns $\emptyset$ because the input s_w is the minimal element of $\prec_w$, and already immediately after the call). Its immediate predecessor is explored when recursing on $(1, 0)$.

Algorithm 1 outputs all sound asynchronizations because after having computed a maximal asynchronization P'_a in a recursive call with parameter s_w, any smaller sound asynchronization is smaller than a predecessor in $\mathsf{ImPred}(P'_a, s_w)$. Also, it can not output the same asynchonization twice. Let P^1_a and P^2_a be two predecessors in $\mathsf{ImPred}(P'_a, s_w)$ obtained by moving up the awaits s^1_w and s^2_w, respectively, and assume that $s^1_w \prec_w s^2_w$. Then, all solutions computed in the recursive call on P^1_a will have s^2_w placed as in P'_a while all the solutions computed in the recursive call on P^2_a will have s^2_w closer to the matching call. Therefore, the sets of solutions computed in these two recursion branches are distinct.

Theorem 1. *AsySyn($\mathsf{wkAsy}[P, L, L_a], s_w$) where s_w is the maximal await in $\mathsf{wkAsy}[P, L, L_a]$ w.r.t. $\prec_w$ outputs all sound asynchronizations of $P[L]$ w.r.t. L_a.*

The delay complexity of Algorithm 1 is polynomial time modulo an oracle that returns a maximal asynchronization relative to a given one. In the next section, we show that the latter problem can be reduced in polynomial time to the reachability problem in sequential programs.

5 Computing Maximal Asynchronizations

In this section, we present an implementation of the procedure MaxRel that relies on a reachability oracle. In particular, we first describe an approach for computing the maximal asynchronization relative to a given asynchronization P_a, which can be seen as a way of repairing P_a so that it becomes data-race free. Intuitively, we repeatedly eliminate data races in P_a by moving certain `await` statements closer to the matching calls. The data races in P_a (if any) are enumerated in a certain order that prioritizes data races between actions that occur first in executions of the original synchronous program. This order allows to avoid superfluous repair steps.

5.1 Data Race Ordering

An action a representing a read/write access in a trace τ of an asynchronization P_a of P is *synchronously reachable* if there is an action a' in a trace τ' of P that represents the same statement, i.e., $\mathsf{S}(a) = \mathsf{S}(a')$. It can be proved that any trace of an asynchronization contains a data race if it contains a data race between two synchronously reachable actions (see Appendix C in [3]). In the following, we focus on data races between actions that are synchronously reachable.

We define an order between such data races based on the order between actions in executions of the original synchronous program P. This order relates data races in possibly different executions or asynchronizations of P, which is possible because each action in a data race corresponds to a statement in P.

For two read/write statements s and s', $s \prec s'$ denotes the fact that there is an execution of P in which the *first* time s is executed occurs before the *first* time s' is executed. For two actions a and a' in an execution/trace of an asynchronization, generated by two read/write statements $s = \mathsf{S}(a)$ and $s' = \mathsf{S}(a')$, $a \prec_{\mathsf{SO}} a'$ holds if $s \prec s'$ and either $s' \not\prec s$ or s' is reachable from s in the interprocedural[2] control-flow graph of P without taking any back edge[3]. For a *deterministic* synchronous program (admitting a single execution), $a \prec_{\mathsf{SO}} a'$ iff $\mathsf{S}(a) \prec \mathsf{S}(a')$. For non-deterministic programs, when $\mathsf{S}(a)$ and $\mathsf{S}(a')$ are contained in a loop body, it is possible that $\mathsf{S}(a) \prec \mathsf{S}(a')$ and $\mathsf{S}(a') \prec \mathsf{S}(a)$. In this case, we use the control-flow order to break the tie between a and a'.

The order between data races corresponds to the colexicographic order induced by $\prec_{\mathsf{SO}}$. This is a partial order since actions may originate from different control-flow paths and are incomparable w.r.t. $\prec_{\mathsf{SO}}$.

Definition 2 (Data Race Order). *Given two races (a_1, a_2) and (a_3, a_4) admitted by (possibly different) asynchronizations of a synchronous program P, we have that $(a_1, a_2) \prec_{\mathsf{SO}} (a_3, a_4)$ iff $a_2 \prec_{\mathsf{SO}} a_4$, or $a_2 = a_4$ and $a_1 \prec_{\mathsf{SO}} a_3$.*

Repairing a minimal data race (a_1, a_2) w.r.t. $\prec_{\mathsf{SO}}$ removes any other data race (a_1, a_4) with $(a_2, a_4) \in \mathsf{HB}$ (note that we cannot have $(a_4, a_2) \notin \mathsf{HB}$ since $a_2 \prec_{\mathsf{SO}} a_4$). The repair will enforce that $(a_1, a_2) \in \mathsf{HB}$ which implies that $(a_1, a_4) \in \mathsf{HB}$.

[2] The interprocedural graph is the union of the control-flow graphs of each method along with edges from call sites to entry nodes, and from exit nodes to return sites.

[3] A back edge points to a block that has already been met during a depth-first traversal of the control-flow graph, and corresponds to loops.

5.2 Repairing Data Races

Repairing a data race (a_1, a_2) reduces to modifying the position of a certain `await`. We consider only repairs where `await`s are moved up (closer to the matching call). The "completeness" of this set of repairs follows from the particular order in which we enumerate data races.

Let s_1 and s_2 be the statements generating a_1 and a_2. In general, there exists a method m that (transitively) calls another asynchronous method $m1$ that contains s_1 and before awaiting for $m1$ it (transitively) calls a method $m2$ that executes s_2. This is pictured in Fig. 10. It is also possible that m itself contains s_2 (see the program on the right of Fig. 7). The repair consists in moving the await for $m1$ before the call to $m2$ since this implies that s_1 will always execute before s_2 (and the corresponding actions are related by happens-before).

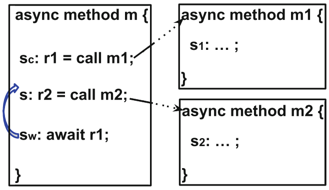

Fig. 10. A data race repair.

Formally, any two racing actions have a common ancestor in the call order CO which is a call action. The least common ancestor of a_1 and a_2 in CO among call actions is denoted by $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$. In Fig. 10, it corresponds to the call statement s_c. More precisely, $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$ is a call action $a_c = (\_, i, \mathsf{call}(j))$ s.t. $(a_c, a_1) \in \mathsf{CO}$, $(a_c, a_2) \in \mathsf{CO}$, and for each other call action a'_c, if $(a_c, a'_c) \in \mathsf{CO}$ then $(a'_c, a_1) \notin \mathsf{CO}$. This call action represents an asynchronous call for which the matching await s_w must move to repair the data race. The await should be moved before the last statement in the same method generating an action which precedes a_2 in the reflexive closure of call order (statement s in Fig. 10). This way every statement that follows s_c in call order will be executed before s and before any statement which succeeds s in call order, including s_2. Note that moving the await s_w anywhere after s will not affect the concurrency between a_1 and a_2.

The pair (s_c, s) is called the *root cause* of the data race (a_1, a_2). We denote by $\mathsf{RDR}(P_a, s_c, s)$ the maximal asynchronization P'_a smaller than P_a w.r.t. $\leq$, s.t. no await statement matching s_c occurs after s on a CFG path.

5.3 A Procedure for Computing Maximal Asynchronizations

Given an asynchronization P_a, the procedure MaxRel in Algorithm 2 computes the maximal asynchronization relative to P_a by repairing data races iteratively until the program becomes data race free. The sub-procedure RCMinDR(P'_a) computes the root cause of a minimal data race (a_1, a_2) of P'_a w.r.t. $\prec_{\mathsf{SO}}$ such that the two actions are synchronously reachable. If P'_a is data race free, then RCMinDR(P'_a) returns $\perp$. The following theorem states the correctness of MaxRel.

Theorem 2. *Given an asynchronization $P_a \in \mathsf{Asy}[P, L, L_a]$, MaxRel$(P_a)$ returns the maximal asynchronization of P relative to P_a.*

Algorithm 2. The procedure MaxRel to find the maximal asynchronization of P relative to P_a.

procedure MaxRel(P_a)
  $P'_a \leftarrow P_a$
  $root \leftarrow$ RCMinDR(P'_a)
  while $root \neq \bot$
    $P'_a \leftarrow \mathsf{RDR}(P'_a, root)$
    $root \leftarrow$ RCMinDR(P'_a)
  return P'_a

MaxRel(P_a) repairs a number of data races which is linear in the size of the input. Indeed, each repair results in moving an await closer to the matching call and before at least one more statement from the original program P.

The problem of computing root causes of minimal data races is reducible to reachability (assertion checking) in sequential programs. This reduction builds on a program instrumentation for checking if there exists a data race that involves two given statements (s_1, s_2) that are reachable in an executions of P. This instrumentation is used in an iterative process where pairs of statements are enumerated according to the colexicographic order induced by $\prec$. For lack of space, we present only the main ideas of the instrumentation (see Appendix D in [3]). The instrumentation simulates executions of an asynchronization P_a using non-deterministic synchronous code where methods may be only partially executed (modeling `await` interruptions). Immediately after executing s_1, the current invocation t_1 is interrupted (by executing a `return` added by the instrumentation). The active invocations that transitively called t_1 are also interrupted when reaching an `await` for an invocation in this call chain (the other invocations are executed until completion as in the synchronous semantics). When reaching s_2, if s_1 has already been executed and at least one invocation has been interrupted, which means that s_1 is concurrent with s_2, then the instrumentation stops with an assertion violation. The instrumentation also computes the root cause of the data race using additional variables for tracking call dependencies.

6 Asymptotic Complexity of Asynchronization Synthesis

We state the complexity of the asynchronization synthesis problem. Algorithm 1 shows that the delay complexity of this problem is polynomial-time in the number of statements in input program modulo the complexity of computing a maximal asynchronization, which Algorithm 2 shows to be polynomial-time reducible to reachability in sequential programs. Since the reachability problem is PSPACE-complete for finite-state sequential programs [16], we get the following:

Theorem 3. *The output complexity*[4] *and delay complexity of the asynchronization synthesis problem is polynomial time modulo an oracle for reachability in sequential programs, and PSPACE for finite-state programs.*

This result is optimal, i.e., checking whether there exists a sound asynchronization which is different from the trivial strong synchronization is PSPACE-hard (follows from a reduction from the reachability problem). See Appendices D and E in [3] for the detailed formal proofs.

7 Asynchronization Synthesis Using Data-Flow Analysis

In this section, we present a refinement of Algorithm 2 that relies on a bottom-up inter-procedural data flow analysis. The analysis is used to compute maximal asynchronizations for abstractions of programs where every Boolean condition (in if-then-else or while statements) is replaced with the non-deterministic choice $*$, and used as an implementation of MaxRel in Algorithm 1.

For a program P, we define an abstraction $P^\#$ where every conditional `if` $\langle le \rangle$ $\{S_1\}$ `else` $\{S_2\}$ is rewritten to `if` $*$ $\{S_1\}$ `else` $\{S_2\}$, and every `while` $\langle le \rangle$ $\{S\}$ is rewritten to `if` $*$ $\{S\}$. Besides adding the non-deterministic choice $*$, loops are unrolled exactly once. Every asynchronization P_a of P corresponds to an abstraction $P_a^\#$ obtained by applying exactly the same rewriting. $P^\#$ is a sound abstraction of P in terms of sound asynchronizations it admits. Unrolling loops once is sound because every asynchronous call in a loop iteration should be awaited for in the same iteration (see the syntactic constraints in Sect. 2).

Theorem 4. *If $P_a^\#$ is a sound asynchronization of $P^\#$ w.r.t. L_a, then P_a is a sound asynchronization of P w.r.t. L_a.*

The procedure for computing maximal asynchronizations of $P^\#$ relative to a given asynchronization $P_a^\#$ traverses methods of $P_a^\#$ in a bottom-up fashion, detects data races using summaries of read/write accesses computed using a straightforward data-flow analysis, and repairs data races using the schema presented in Sect. 5.2. Applying this procedure to a real programming language requires an alias analysis to detect statements that may access the same memory location (this is trivial in our language which is used to simplify the exposition).

We consider an enumeration of methods called *bottom-up order*, which is the reverse of a topological ordering of the call graph[5]. For each method m, let $\mathcal{R}(m)$ be the set of program variables that m can read, which is defined as the union of $\mathcal{R}(m')$ for every method m' called by m and the set of program variables read in statements in the body of m. The set of variables $\mathcal{W}(m)$ that m can write is defined in a similar manner. We define $\mathsf{RW\text{-}var}(m) = (\mathcal{R}(m), \mathcal{W}(m))$. We extend the notation $\mathsf{RW\text{-}var}$ to statements as follows: $\mathsf{RW\text{-}var}(\langle r \rangle := \langle x \rangle) = (\{x\}, \emptyset)$, $\mathsf{RW\text{-}var}(\langle x \rangle := \langle le \rangle) = (\emptyset, \{x\})$, $\mathsf{RW\text{-}var}(r := \texttt{call } m) = \mathsf{RW\text{-}var}(m)$,

[4] Note that all asynchronizations can be enumerated with polynomial space.

[5] The nodes of the call graph are methods and there is an edge from a method m_1 to a method m_2 if m_1 contains a call statement that calls m_2.

and $\mathsf{RW\text{-}var}(s) = (\emptyset, \emptyset)$, for any other type of statement s. Also, let $\mathsf{CRW\text{-}var}(m)$ be the set of read or write accesses that m can do and that can be concurrent with accesses that a caller of m can do after calling m. These correspond to read/write statements that follow an `await` in m, or to accesses in $\mathsf{CRW\text{-}var}(m')$ for a method m' called by m. These sets of accesses can be computed using the following data-flow analysis: for all methods $m \in P_a^{\#}$ in bottom-up order, and for each statement s in the body of m from begin to end,

- if s is a call to m' and s is *not* reachable from an `await` in the CFG of m
  - $\mathsf{CRW\text{-}var}(m) \leftarrow \mathsf{CRW\text{-}var}(m) \cup \mathsf{CRW\text{-}var}(m')$
- if s is reachable from an `await` statement in the CFG of m
  - $\mathsf{CRW\text{-}var}(m) \leftarrow \mathsf{CRW\text{-}var}(m) \cup \mathsf{RW\text{-}var}(s)$

We use $(\mathcal{R}_1, \mathcal{W}_1) \bowtie (\mathcal{R}_2, \mathcal{W}_2)$ to denote the fact that $\mathcal{W}_1 \cap (\mathcal{R}_2 \cup \mathcal{W}_2) \neq \emptyset$ or $\mathcal{W}_2 \cap (\mathcal{R}_1 \cup \mathcal{W}_1) \neq \emptyset$ (i.e., a conflict between read/write accesses). We define the procedure $\textsc{MaxRel}^{\#}$ that given an asynchronization $P_a^{\#}$ works as follows:

- for all methods $m \in P_a^{\#}$ in bottom-up order, and for each statement s in the body of m from begin to end,
  - if s occurs between r := `call` m' and `await` r (for some m'), and $\mathsf{RW\text{-}var}(s) \bowtie \mathsf{CRW\text{-}var}(m')$, then $P_a^{\#} \leftarrow \mathsf{RDR}(P_a^{\#}, r := \texttt{call}\ m', s)$
- return $P_a^{\#}$

Theorem 5. *The procedure* $\textsc{MaxRel}^{\#}(P_a^{\#})$ *returns a maximal asynchronization relative to* $P_a^{\#}$*.*

Since $\textsc{MaxRel}^{\#}$ is based on a single bottom-up traversal of the call graph of the input asynchronization $P_a^{\#}$ we get the following result.

Theorem 6. *The delay complexity of the asynchronization synthesis problem restricted to abstracted programs* $P^{\#}$ *is polynomial time.*

8 Experimental Evaluation

We present an empirical evaluation of our asynchronization synthesis approach, where maximal asynchronizations are computed using the data-flow analysis in Sect. 7. Our benchmark consists mostly of asynchronous C# programs from open-source GitHub projects. We evaluate the effectiveness in reproducing the original program as an asynchronization of a program where asynchronous calls are reverted to synchronous calls, along with other sound asynchronizations.

Implementation. We developed a prototype tool that uses the Roslyn .NET compiler platform [27] to construct CFGs for methods in a C# program. This prototype supports C# programs written in static single assignment (SSA) form that include basic conditional/looping constructs and async/await as concurrency primitives. Note that object fields are interpreted as program variables in the terminology of Sect. 2 (data races concern accesses to object fields). It assumes that alias information is provided apriori; these constraints can be removed in the future with more engineering effort. In general, our synthesis procedure is compatible with any sound alias analysis. The precision of this analysis impacts only the set (number) of asynchronizations outputted by the procedure (a more precise analysis may lead to more sound asynchronizations).

The tool takes as input a possibly asynchronous program, and a mapping between synchronous and asynchronous variations of base methods in this program. It reverts every asynchronous call to a synchronous call, and it enumerates sound asynchronizations of the obtained program (using Algorithm 1).

Benchmark. Our evaluation uses a benchmark listed in Table 2, which contains 5 synthetic examples (variations of the program in Fig. 1), 9 programs extracted from open-source C# GitHub projects (their name is a prefix of the repository name), and 2 programs inspired by questions on `stackoverflow.com` about async/await in C# (their name ends in Stackoverflow). Overall, there are 13 base methods involved in computing asynchronizations of these programs (having both synchronous and asynchronous versions), coming from 5 C#

Table 2. Empirical results. Syntactic characteristics of input programs: lines of code (loc), number of methods (m), number of method calls (c), number of asynchronous calls (ac), number of awaits that *could* be placed at least one statement away from the matching call ($\text{await}_{\#}$). Data concerning the enumeration of asynchronizations: number of awaits that *were* placed at least one statement away from the matching call (await), number of races discovered and repaired (races), number of statements that the awaits in the maximal asynchronization are covering *more than* in the input program (cover), number of computed asynchronizations (async), and running time (t).

Program	loc	m	c	ac	$\text{await}_{\#}$	await	races	cover	async	t(s)
SyntheticBenchmark-1	77	3	6	5	4	4	5	0	9	1.4
SyntheticBenchmark-2	115	4	12	10	6	3	3	0	8	1.4
SyntheticBenchmark-3	168	6	16	13	9	7	4	0	128	1.5
SyntheticBenchmark-4	171	6	17	14	10	8	5	0	256	1.9
SyntheticBenchmark-5	170	6	17	14	10	8	9	0	272	2
Azure-Remote	520	10	14	5	0	0	0	0	1	2.2
Azure-Webjobs	190	6	14	6	1	1	0	1	3	1.6
FritzDectCore	141	7	11	8	1	1	0	1	2	1.6
MultiPlatform	53	2	6	4	2	2	0	2	4	1.1
NetRpc	887	13	18	11	4	1	3	0	3	2
Scoreboards	43	3	3	3	0	0	0	0	1	1.5
VBForums-Viewer	275	7	10	7	3	2	1	1	6	1.8
Voat	178	3	5	5	2	1	1	1	3	1.2
WordpressRESTClient	133	3	10	8	4	2	1	0	4	1.7
ReadFile-Stackoverflow	47	2	3	3	1	0	1	0	1	1.5
UI-Stackoverflow	50	3	4	4	3	3	3	0	12	1.5

libraries (*System.IO*, *System.Net*, *Windows.Storage*, *Microsoft.WindowsAzure.-Storage*, and *Microsoft.Azure.Devices*). They are modeled as described in Sect. 2.

Evaluation. The last five columns of Table 2 list data concerning the application of our tool. The column async lists the number of outputted sound asynchronizations. In general, the number of asynchronizations depends on the number of invocations (column ac) and the size of the code blocks between an invocation and the instruction using its return value (column $\text{await}_{\#}$ gives the number of non-empty blocks). The number of *sound* asynchronizations depends roughly, on how many of these code blocks are racing with the method body. These asynchronizations contain `await`s that are at a non-zero distance from the matching call (non-zero values in column await) and for many Github programs, this distance is bigger than in the original program (non-zero values in column cover). This shows that we are able to increase the distances between `await`s and their matching calls for those programs. The distance between `await`s and matching calls in maximal asynchronizations of non synthetic benchmarks is 1.27 statements on average. A statement representing a method call is counted as one independently of the method's body size. With a single level of inlining, the number of statements becomes 2.82 on average. However, these statements are again, mostly IO calls (access to network or disk) or library calls (string/bytes formatting methods) whose execution time is not negligible. The running times for the last three synthetic benchmarks show that our procedure is scalable when programs have a large number of sound asynchronizations.

With few exceptions, each program admits multiple sound asynchronizations (values in column async bigger than one), which makes the focus on the delay complexity relevant. This leaves the possibility of making a choice based on other criteria, e.g., performance metrics. As shown by the examples in Fig. 2, their performance can be derived only dynamically (by executing them). These results show that our techniques have the potential of becoming the basis of a refactoring tool allowing programmers to improve their usage of the async/await primitives. The artifact is available at [2].

9 Related Work

There are many works on synthesizing or repairing concurrent programs in the standard multi-threading model, e.g., automatic parallelization in compilers [1, 7, 19], or synchronization synthesis [6, 10–12, 18, 24, 30, 31]. We focus on the use of async/await which poses specific challenges not covered in these works.

Our semantics without `await` $*$ instructions is equivalent to the semantics defined in [4, 28]. But, to simplify the exposition, we consider a more restricted programming language. For the modeling of asynchronous IO operations, we follow [4] with the restriction that the code following an `await` $*$ is executed atomically. This is sound when focusing on data-race freedom because even if executed atomically, any two instructions from different asynchronous IO operations (following `await` $*$) are not happens-before related.

Program Refactoring. Program refactoring tools have been proposed for converting C# programs using explicit callbacks into async/await programs [25] or Android programs using AsyncTask into programs that use IntentService [22]. The C# tool [25], which is the closest to our work, makes it possible to repair misusage of async/await that might result in deadlocks. This tool cannot modify procedure calls to be asynchronous as in our work. A static analysis based technique for refactoring JavaScript programs is proposed in [17]. As opposed to our work, this refactoring technique is unsound in general. It requires that programmers review the refactoring for correctness, which is error-prone. Also, in comparison to [17], we carry a formal study of the more general problem of finding all sound asynchronizations and investigate its complexity.

Data Race Detection. Many works study dynamic data race detection using happens-before and lock-set analysis, or timing-based detection [14,20,21,26,29]. They could be used to approximate our reduction from data race checking to reachability in sequential programs. Some works [5,13,23] propose static analyses for finding data races. [5] designs a compositional data race detector for multi-threaded Java programs, based on an inter-procedural analysis assuming that any two public methods can execute in parallel. Similar to [28], they precompute method summaries to extract potential racy accesses. These approaches are similar to the analysis in Sect. 7, but they concern a different programming model.

Analyzing Asynchronous Programs. Several works propose program analyses for various classes of asynchronous programs. [8,15] give complexity results for the reachability problem, and [28] proposes a static analysis for deadlock detection in C# programs that use both asynchronous and synchronous wait primitives. [9] investigates the problem of checking whether Java UI asynchronous programs have the same set of behaviors as sequential programs where roughly, asynchronous tasks are executed synchronously.

10 Conclusion

We proposed a framework for refactoring sequential programs to equivalent asynchronous programs based on async/await. We determined precise complexity bounds for the problem of computing all sound asynchronizations. This problem makes it possible to compute a sound asynchronization that maximizes performance by separating concerns – enumerate sound asynchronizations and evaluate performance separately. On the practical side, we have introduced an approximated synthesis procedure based on data-flow analysis that we implemented and evaluated on a benchmark of non-trivial C# programs.

The asynchronous programs rely exclusively on async/await and are deadlock-free by definition. Deadlocks can occur in a mix of async/await with "explicit" multi-threading that includes blocking `wait` primitives. Extending our approach for such programs is an interesting direction for future work.

References

1. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4), 345–420 (1994). https://doi.org/10.1145/197405.197406
2. Beillahi, S.M., Bouajjani, A., Enea, C., Lahiri, S.: Artifact for the SAS 2022 paper: Automated Synthesis of Asynchronizations (May 2022). DOI: https://doi.org/10.5281/zenodo.7055422,https://doi.org/10.5281/zenodo.7055422
3. Beillahi, S.M., Bouajjani, A., Enea, C., Lahiri, S.: Automated synthesis of asynchronizations. CoRR abs/2209.06648 (2022), https://arxiv.org/abs/2209.06648
4. Bierman, G., Russo, C., Mainland, G., Meijer, E., Torgersen, M.: Pause n play: formalizing asynchronous C$^\sharp$. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 233–257. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31057-7_12
5. Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: Racerd: compositional static race detection. Proc. ACM Program. Lang. **2**(OOPSLA), 144:1–144:28 (2018). https://doi.org/10.1145/3276514
6. Bloem, R., Hofferek, G., Könighofer, B., Könighofer, R., Ausserlechner, S., Spork, R.: Synthesis of synchronization using uninterpreted functions. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014. pp. 35–42. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987593
7. Blume, W., et al.: Parallel programming with Polaris. Computer **29**(12), 81–87 (1996). https://doi.org/10.1109/2.546612
8. Bouajjani, A., Emmi, M.: Analysis of recursively parallel programs. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, 22–28 January 2012, pp. 203–214. ACM (2012). https://doi.org/10.1145/2103656.2103681
9. Bouajjani, A., Emmi, M., Enea, C., Ozkan, B.K., Tasiran, S.: Verifying robustness of event-driven asynchronous programs against concurrency. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 170–200. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_7
10. Černý, P., Clarke, E.M., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Samanta, R., Tarrach, T.: From Non-preemptive to preemptive scheduling using synchronization synthesis. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 180–197. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_11
11. Černý, P., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Tarrach, T.: Regression-free synthesis for concurrency. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 568–584. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_38
12. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
13. Engler, D.R., Ashcraft, K.: Racerx: effective, static detection of race conditions and deadlocks. In: Scott, M.L., Peterson, L.L. (eds.) Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, 19–22 October 2003, pp. 237–252. ACM (2003). https://doi.org/10.1145/945445.945468

14. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, 15–21 June 2009, pp. 121–133. ACM (2009). https://doi.org/10.1145/1542476.1542490
15. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. **34**(1), 6:1–6:48 (2012). https://doi.org/10.1145/2160910.2160915
16. Godefroid, P., Yannakakis, M.: Analysis of Boolean programs. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 214–229. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_16
17. Gokhale, S., Turcotte, A., Tip, F.: Automatic migration from synchronous to asynchronous javascript apis. Proc. ACM Program. Lang. **5**(OOPSLA), 1–27 (2021). https://doi.org/10.1145/3485537
18. Gupta, A., Henzinger, T.A., Radhakrishna, A., Samanta, R., Tarrach, T.: Succinct representation of concurrent trace sets. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 433–444. ACM (2015). https://doi.org/10.1145/2676726.2677008
19. Han, H., Tseng, C.: A comparison of parallelization techniques for irregular reductions. In: Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, 23–27 April 2001, p. 27. IEEE Computer Society (2001). https://doi.org/10.1109/IPDPS.2001.924963
20. Kini, D., Mathur, U., Viswanathan, M.: Dynamic race prediction in linear time. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 157–170. ACM (2017). https://doi.org/10.1145/3062341.3062374
21. Li, G., Lu, S., Musuvathi, M., Nath, S., Padhye, R.: Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, 27–30 October 2019. pp. 162–180. ACM (2019). https://doi.org/10.1145/3341301.3359638
22. Lin, Y., Okur, S., Dig, D.: Study and refactoring of android asynchronous programming (T). In: Cohen, M.B., Grunske, L., Whalen, M. (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, 9–13 November 2015, pp. 224–235. IEEE Computer Society (2015). https://doi.org/10.1109/ASE.2015.50
23. Liu, B., Huang, J.: D4: fast concurrency debugging with parallel differential analysis. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018, pp. 359–373. ACM (2018). https://doi.org/10.1145/3192366.3192390
24. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. **6**(1), 68–93 (1984). https://doi.org/10.1145/357233.357237
25. Okur, S., Hartveld, D.L., Dig, D., van Deursen, A.: A study and toolkit for asynchronous programming in c#. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India - May 31–June 07, 2014, pp. 1117–1127. ACM (2014). https://doi.org/10.1145/2568225.2568309

26. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Efficient data race detection for Async-finish parallelism. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 368–383. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_28
27. Roslyn: (2022). https://github.com/dotnet/roslyn
28. Santhiar, A., Kanade, A.: Static deadlock detection for asynchronous c# programs. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 292–305. ACM (2017). https://doi.org/10.1145/3062341.3062361
29. Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound predictive race detection in polynomial time. In: Field, J., Hicks, M. (eds.) Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, 22–28 January 2012, pp. 387–400. ACM (2012). https://doi.org/10.1145/2103656.2103702
30. Vechev, M., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 139–154. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_13
31. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 327–338. ACM (2010). https://doi.org/10.1145/1706299.1706338