Automated Synthesis of Asynchronizations

SIDI MOHAMED BEILLAHI, Université de Paris, France
AHMED BOUAJJANI, Université de Paris, France
CONSTANTIN ENEA, Université de Paris, France
SHUVENDU LAHIRI, Microsoft Research Lab - Redmond, USA

Asynchronous programming is widely adopted for building responsive and efficient software. Modern languages such as C# provide async/await primitives to simplify the use of asynchrony. However, the use of these primitives remains error-prone because of the non-determinism in their semantics. In this paper, we propose an approach for refactoring a given 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 given asynchronous versions, and it is constrained to avoid introducing data races. Since the space of possible solutions is exponential in general, we focus on characterizing the delay complexity that quantifies the delay between two consecutive and distinct outputs. We show that this is polynomial time modulo an oracle for solving reachability (assertion checking) in sequential programs. We also describe a pragmatic approach based on an interprocedural data-flow 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. Unlike synchronous procedure calls, asynchronous procedure calls may run only partially and return the control to their caller. Later, when the callee finishes execution, a callback procedure registered by the caller is invoked.

As an alternative to the tedious model of asynchronous programming that required explicitly registering callbacks with asynchronous calls, C# 5.0 [Bierman et al. 2012] 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 by the keyword `async`, returns a task object that the caller uses to "await" it. Awaiting may suspend the execution of the caller, if the awaited task did not finish, but does not block the thread it is running on. The code after the await is the continuation that is automatically called back when the callee result is ready. For instance, on the right of Figure 1, the method `ContentLength` calls an asynchronous method `GetStringAsync` that returns a task object `t5` used to await it at line 24. Executing this await suspends the execution of `ContentLength` and returns the control to its caller `MainAsync`, assuming that `GetStringAsync` did not finish. Passing the control to the caller is a constraint of the await semantics. When `GetStringAsync` finishes and the thread is idle, the continuation after line 24 is scheduled. This paradigm has become popular across many languages, eg, C++, JavaScript, Python.

While simplifying the writing of asynchronous programs, the async/await primitives introduce concurrency which is notoriously complex. Depending on the scheduler, the code in between a call and a matching await (referring to the same task) may execute before some part of the awaited task (if the latter passed the control to its caller before finishing), or after the awaited task finished. For instance, on the right of Figure 1, the assignment `x=0` in `MainAsync` between the call and the await of `ReadFile` may execute before `ReadFile` finishes, if the `await` in `ReadFile` suspends

Authors' addresses: Sidi Mohamed Beillahi, Université de Paris, France, beillahi@irif.fr; Ahmed Bouajjani, Université de Paris, France, abou@irif.fr; Constantin Enea, Université de Paris, France, cenea@irif.fr; Shuvendu Lahiri, Microsoft Research Lab - Redmond, USA, shuvendu@microsoft.com.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

```
1 void Main(string url0) {
2  string url1 = ReadFile("url1.txt");
3  x = 0;
4  int val0 = ContentLength(url0);


7  int val1 = ContentLength(url1);

9  int r = x;
10 Debug.Assert(r == val0 + val1);
11 }

13 string ReadFile(string fn) {
14  StreamReader reader = new StreamReader(fn);
15  string content = reader.ReadToEnd();

17  return content;
18 }

20 int ContentLength(string url) {
21  HttpClient clt = new HttpClient()
22  string urlContents = clt.GetString(url);
23  int r1 = x;

25  x = r1 + urlContents.Length;
26  return urlContents.Length;
27 }
```

```
1 async Task MainAsync(string url0) {
2  Task<string> t1 = ReadFile("url1.txt");
3  x = 0;
4  Task<int> t2 = ContentLength(url0);
5  string url1 = await t1;
6  int val0 = await t2;
7  Task<int> t3 = ContentLength(url1);
8  int val1 = await t3;
9  int r = x;
10 Debug.Assert(r == val0 + val1);
11 }

13 async Task<string> ReadFile(string fn) {
14  StreamReader reader = new StreamReader(fn);
15  Task<string> t4 = reader.ReadToEndAsync();
16  string content = await t4;
17  return content;
18 }

20 async Task<int> ContentLength(string url) {
21  HttpClient clt = new HttpClient();
22  Task<string> t5 = clt.GetStringAsync(url);
23  int r1 = x;
24  urlContents = await t5;
25  x = r1 + urlContents.Length;
26  return urlContents.Length;
27 }
```

Fig. 1. A synchronous C# program and an asynchronous refactoring (x is a static variable).

its execution and passes the control to `MainAsync` (when reaching the await at line 16 because `ReadToEndAsync` did not finish), or after `ReadFile` finishes, otherwise (the call to `ReadToEndAsync` finishes before reaching its matching await at line 16). The resemblance with sequential code can be especially deceitful since this non-determinism is opaque. It is common that await instructions are placed immediately after the corresponding call which limits the benefits that one can obtain from executing code in the caller concurrently with code in the callee [Okur et al. 2014].

In this paper, we address the problem of writing efficient asynchronous code that uses async/await primitives. 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. Since solving this problem in its full generality would require checking equivalence between arbitrary programs, which is known to be hard, we consider a restricted space of asynchronous program candidates that are defined by substituting synchronous methods in P with asynchronous versions (assumed to be behaviorally equivalent). The substituted methods are supposed to be leaves of the call-tree, i.e., they do not call any other method in P. Such programs are called *asynchronizations* of P. A practical instantiation of this problem is replacing IO synchronous calls for, e.g., reading/writing files, managing http connections, with asynchronous versions.

For instance, let us consider the sequential C# program on the left of Fig. 1. The `Main` invokes `ReadFile` and `ContentLength` in a synchronous way (using the standard call stack semantics). `ReadFile` reads and returns the content of a file while `ContentLength` returns the length of the text in a webpage. The two URLs given as input to `ContentLength` are given as input to `Main` or read from some file using `ReadFile`. The program uses a variable `x` to aggregate the lengths of all pages accessed by `ContentLength`. Note that this program passes the assertion at line 10.

The time-consuming primitives for reading files, `StreamReader.ReadToEnd`, or the content of a webpage, `HttpClient.GetString`[1], are an obvious choice for being replaced with equivalent *asynchronous* counterparts, i.e., `StreamReader.ReadToEndAsync` and `HttpClient.GetStringAsync`, respectively. Performing such tasks asynchronously can lead to significant boosts in performance.

The program on the right of Fig. 1 is an example of an asynchronization of the program on the left where the calls to `StreamReader.ReadToEnd` and `HttpClient.GetString` are replaced with asynchronous counterparts (assumed to have the same effect). The syntax of async/await imposes that every method that transitively calls one of the substituted methods must also be declared to be asynchronous, e.g., `MainAsync` and `ContentLength`. Then, an asynchronous call must be followed by an `await` statement that specifies the control location where that task should have completed (e.g., the return value should have been computed). For instance, the call to `ReadToEndAsync` at line 15 is immediately followed by an `await` since the next instruction (at line 17) uses the value computed by `ReadToEndAsync`. Therefore, synthesizing such refactoring boils down to finding a correct placement of `await`s for every 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 instead of a more precise equivalence relation like equality of reachable sets of states could prevent enumerating some number of asynchronizations that reach the same set of states as the original synchronous program. However, checking equality of reachable sets of states is known to be hard in general, and relying on absence of data races is a well established compromise. For instance, the asynchronization on the right of Fig. 1 is sound (data-race free) since the two calls to `ContentLength` that access `x` do not "overlap" and both finish before the read at line 9. The accesses to `x` in the asynchronization are performed in the same order as in the original 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. For instance, the `await` at line 24 can be moved one statement up (before the read of `x`) and the resulting program remains equivalent to the sequential one. Thus, we consider the problem of enumerating *all* sound asynchronizations of a sequential program P w.r.t. substituting a set of methods with asynchronous versions. Enumerating all sound asynchronizations 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). This problem reduces to finding all possible placements of awaits that do not introduce data races.

In general, the number of (sound) asynchronizations is exponential in the number of method calls in the program. Therefore, we focus on the *delay* complexity of this enumeration problem, i.e., the complexity of the delay between outputting two consecutive (distinct) outputs. Note that a trivial enumeration of all asynchronizations and checking equivalence to the original program for each one of them has an exponential delay complexity. We also consider the problem of computing *optimal* asynchronizations that maximize the distance between a call and a matching await. The code in between these two statements can execute in parallel with the awaited task, and therefore, optimal asynchronizations maximize the amount of parallelism. Note however that it is hard to argue that such maximal parallelism translates always to maximal performance in practice.

We show that both the delay complexity of the enumeration problem, and the complexity of computing an optimal asynchronization are polynomial time modulo an oracle for solving reachability (assertion checking) in *sequential* programs (they both reduce to a quadratic number

[1] Actually, the `.Net` platform does not contain such a method. We use it here to simplify the exposition. Reading the content of a webpage should pass through `WebRequest` and `HttpWebResponse` objects. The explanations would remain valid.

of reachability queries). The former relies on the latter via a rather surprising result, which differs from other concurrency synthesis problems (e.g., insertion of locks), which is that the optimal asynchronization is *unique*. This holds even if the optimality is relative to a given asynchronization P_a which intuitively, imposes an upper bound on the distance between awaits and matching calls. In general, one could expect that 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 essentially, the optimal asynchronization is required to be equivalent to a *sequential* program, which is deterministic and executes statements in a fixed order. To show the robustness of these results, we also investigate the related problem of synthesizing sound multi-threaded refactorings, where every method call is executed by a different thread. We show that the techniques used to compute asynchronizations can be extended to this case as well.

As a more pragmatic approach, we define a procedure for computing sound asynchronizations which relies on a bottom-up interprocedural data-flow analysis. Intuitively, the placement of awaits is computed by traversing the call graph bottom up, from "base" methods that do not call any other method in the program, to methods that call only base methods, and so on. Each method m is considered only once, and the placement of awaits in m is derived based on a data-flow analysis that computes read or write accesses made in the callees. We show that this procedure computes optimal asynchronizations of abstracted programs where every Boolean condition in if-then-else constructs or while loops is replaced with non-deterministic choice. These asynchronizations are sound for the concrete programs as well. This procedure enables a polynomial delay enumeration of the 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. This evaluation shows that sound asynchronizations can be enumerated efficiently and in some cases, we found asynchronizations that increase the amount of parallelism (the distance between calls and awaits). This demonstrates 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:

- We define the problem of data race-free asynchronization synthesis for refactoring sequential code to equivalent asynchronous code.
- We show that the optimization problem of computing a data race-free asynchronization that maximizes the distance between calls and awaits admits a unique solution.
- We investigate the delay complexity of data race-free asynchronization synthesis.
- A pragmatic algorithm based on data-flow analysis for computing asynchronizations.
- A prototype implementation of this algorithm and an evaluation of this prototype on a benchmark of non-trivial C# programs extracted from open-source repositories.

2 OVERVIEW

We give an overview of our techniques for synthesizing sound asynchronizations using as example the synchronous program on the left of Fig. 2. We discuss asynchronizations obtained by replacing the calls to `IO` with equivalent asynchronous counterparts `IOAsync`. The program on the center of Fig. 2 is the "weakest" asynchronization where the awaits cannot be moved further away from their matching calls because of the use of the return values. The methods `MainAsync`, `F1`, and `F2` are declared to be asynchronous since they (in)directly call `IOAsync`. This program is not a solution to our synthesis problem since it is not equivalent to the sequential program. It admits new behaviors, an example being pictured in Fig. 3 (edges represent execution order): the accesses to `x` and `y` in `MainAsync` occur before the write to `x` in `F1` and the write to `y` in `F2`, respectively. These statements

```
1 void Main() {
2  F1();
3  F2();

5  int r1 = x;

7  int r2 = y;
8  x = r1 + r2;

11 }
12 void F1() {
13  int r3 = IO();

15  x = r3;
16 }
17 void F2() {
18  int r4 = IO();

20  y = r4;
21 }
```

```
1 async Task MainAsync() {
2  Task t1 = F1();
3  Task t2 = F2();

5  int r1 = x;

7  int r2 = y;
8  x = r1 + r2;
9  await t1;
10 await t2;
11 }
12 async Task F1() {
13  Task<int> t3 = IOAsync();
14  int r3 = await t3;
15  x = r3;
16 }
17 async Task F2() {
18  Task<int> t4 = IOAsync();
19  int r4 = await t4;
20  y = r4;
21 }
```

```
1 async Task MainAsync() {
2  Task t1 = F1();
3  Task t2 = F2();
4  await t1;
5  int r1 = x;
6  await t2;
7  int r2 = y;
8  x = r1 + r2;

11 }
12 async Task F1() {
13  Task<int> t3 = IOAsync();
14  int r3 = await t3;
15  x = r3;
16 }
17 async Task F2() {
18  Task<int> t4 = IOAsync();
19  int r4 = await t4;
20  y = r4;
21 }
```

Fig. 2. A synchronous C# program and two asynchronizations (x and y are static variables).

were executed in the opposite order in the sequential program. They form three data races because they are not ordered by the control-flow of the asynchronization. There is another execution where they execute as in the original program: if tasks `t3` and `t4` finish immediately, then the `await` has no effect, and `F1` and `F2` finish before returning control to their caller `MainAsync`.

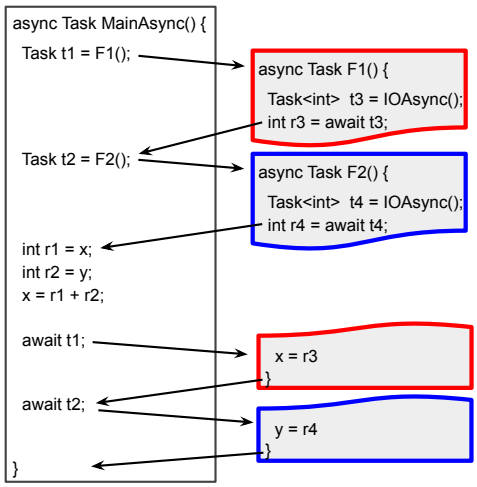

Fig. 3. An execution with three data races on x and y. The blocks of instructions executed in F1 and F2 are marked with red and blue outlines, respectively. Each call is decomposed into two blocks representing what is executed before they are suspended (due to an await of IOAsync) and when the continuation is scheduled.

We define a procedure for computing a data race-free asynchronization, which is optimal in the sense that it maximizes the distance between calls and matching awaits. This is an iterative process that repairs data races starting from the "weakest" asynchronization on the center of Fig. 2. For instance, the data race between the write to x in `MainAsync` and the write to x in `F1` in Fig. 3 can be repaired by moving the `await t1` one position up, before the the write to x in `MainAsync`. This way, the write to x in `F1` will always execute first. The call to `F1` that matches this await and the write to x in `MainAsync` are regarded as the *root cause* of this data race.

For efficiency, the data races are enumerated and repaired in a certain order, that avoids superfluous repair steps. This order prioritizes data races involving statements that would execute first in the original sequential program. For instance, in Fig. 3, the first data race to repair involves the read of x from `MainAsync` and the write to x in `F1`, because these statements are the first to execute in the original sequential program among the other statements involved in data races. Repairing this data race consists in moving `await t1` before the read of x from `MainAsync`, which implies that `F1` completes before the read of x. This repair is defined from a notion of *root cause* of a data race, that in this case, contains the call to `F1` and the read of x from `MainAsync`. Interestingly, this repair step removes

the write-write data race between the write to x in MainAsync and the write to x in F1 as well. If we would have repaired these data races in the opposite order, we would have moved `await t1` first before the write to x, and then, before the read of x. Similarly, the data race between the read of y from MainAsync and the write to y in F2 is repaired by moving the `await t2` before the the read from y in MainAsync (the call to F2 and the read from y in MainAsync are the *root cause* of this data race). The resulting program is shown on the right of Fig. 2 and it is equivalent to the original sequential program. We show that the problem of computing root-causes of data races which are minimal in this order can be reduced in polynomial time to reachability (assertion checking) in sequential programs.

Asynchronizations can be partially ordered depending on the distance, i.e., the number of statements from the original program, between a call and a matching await. The left of Fig. 4 pictures an excerpt of this partial order where an asynchronization is represented as a vector of distances, the first element is the number of statements between the call and the await on t1 (we count only statements that appear in the sequential program as well and exclude awaits), and so on. The asynchronization on the right of Fig. 1 corresponds to $(1, 1, 0, 0)$. The bottom of this order represents an asynchronization where every call is immediately followed by await, and it has the same semantics as the original program. The top element is the "weakest" asynchronization, which was given in the middle of Fig. 1. The right of Fig. 4 gives the set of all sound asynchronizations. The program on the right of Fig. 1 is the biggest element, i.e., moving any `await` further away from the matching call introduces a data race.

To enumerate all sound asynchronizations, we perform a top-down traversal of the partial order on the left of Fig. 4. We first compute the biggest element that is data race free. Although this is a partial order, we show that this element is actually *unique*. Intuitively, uniqueness is proved by contradiction, showing that the least upper bound of two maximal incomparable sound asynchronizations is also a sound asynchronization. For instance, the least common ancestor of the two sound asynchronizations with vectors of distances $(1, 0, 0, 0)$ and $(0, 1, 0, 0)$ is the sound asynchronization $(1, 1, 0, 0)$.

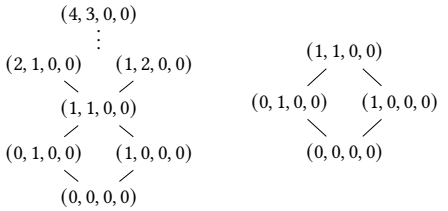

Fig. 4. Partially-ordered sets of asynchronizations of the program on the left of Fig. 2. The edges connect comparable elements, smaller elements being below bigger elements.

Then, for each immediate predecessor P_a of the biggest sound asynchronization, we compute the biggest sound asynchronization which is smaller than P_a (the first step is a particular case where P_a is the top element). As an extension of the previous case, this is also unique, and called an optimal asynchronization relative to P_a. Ensuring that traversals starting in different immediate predecessors explore disjoint parts of the asynchonization space requires some additional constraints on the exploration, which are explained in Section 5. The enumeration finishes when reaching the bottom, which is data race free by definition, on all branches of the recursion.

As a pragmatic alternative, we propose a procedure based on static analysis for enumerating sound asynchronizations, which follows essentially the same schema, except that the problem of data race detection is delegated to a static analysis.

3 ASYNCHRONOUS PROGRAMS

Fig. 5 lists the syntax of a simple programming language used to formalize our approach. A *program* is defined by set of methods, including a distinguished `main`, which are classified as *synchronous* or *asynchronous*. Synchronous methods execute immediately as they are invoked and run continuously until completion. Asynchronous methods, marked using the keyword `async`, can run only partially

$\langle prog \rangle$ ::= `program` $\langle md \rangle$

$\langle md \rangle$ ::= `method` $\langle m \rangle$ $\langle inst \rangle$ | `async method` $\langle m \rangle$ $\langle inst \rangle$ | $\langle md \rangle$; $\langle md \rangle$

$\langle inst \rangle$ ::= $\langle x \rangle$:= $\langle le \rangle$ | $\langle r \rangle$:= $\langle x \rangle$ | $\langle r \rangle$:= `call` $\langle m \rangle$ | `return` | `await` $\langle r \rangle$ | `await` $*$ | `if` $\langle le \rangle$ {$\langle inst \rangle$} `else` {$\langle inst \rangle$}
| `while` $\langle le \rangle$ {$\langle inst \rangle$} | $\langle inst \rangle$; $\langle inst \rangle$

Fig. 5. 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.

and be interrupted when executing an `await`. Only asynchronous methods are allowed to 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 consists of a method name from a set $\mathbb{M}$ and a method body, i.e., a list of statements. These statements use 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,...). We assume that input/return parameters are modeled using dedicated program variables. We assume that each method call returns a *unique task identifier* from a set $\mathbb{T}$, which is used to record control dependencies imposed by `await`s (for uniformity, synchronous methods return a task identifier as well). Our language includes assignments to local/program variables, `await`s, `return` statements, `while` loops, and conditionals. We assume that variables take values from a data domain $\mathbb{D}$, which includes $\mathbb{T}$ to account for variables storing task identifiers. The assignment to a local variable $\langle r \rangle := \langle x \rangle$, where x is a program variable, is called a *read* of $\langle x \rangle$ and an assignment to a program variable $\langle x \rangle := \langle le \rangle$ is called a *write* to $\langle x \rangle$. 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).

For example, Fig. 6 lists the modeling in our language of IO methods `ReadToEndAsync` and `GetStringAsync` used in Fig. 1. We use program variables to represent system resources such as the network or 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. `GetStringAsync` contains a read of the resource `WWW` (for world wide web) at some input url. 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.

```
async method GetStringAsync() {
  await *;
  retVal  = WWW[url_Input];
  return
}
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. 6. Modeling IO operations.

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` ... (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` ... 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 which are not caught. In this work, we forbid passing task identifiers as method parameters or return values (which is possible in C#). An `await` r statement is said to *match* an r := `call` m' statement.

For example, the program on the left of Fig. 7 does not satisfy the second condition above since `await r` can be reached without entering the loop. The program in the center of Fig. 7 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. 7 satisfies both conditions.

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

  await r;
}
```

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

  if *
    await r;
}
```

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

Fig. 7. Examples of programs

Semantics. A program configuration is a tuple (g, stack, pending, completed, c-by, w-for) where g is composed of the valuation of the program variables, stack is the call stack, pending is the set of asynchronous tasks, e.g., continuations predicated on the completion of some method call, completed is the set of completed tasks, c-by represents the relation between a method call and its caller, and w-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 completed : $\mathbb{T} \to \{\top, \bot\}$ such that completed$(i) = \top$ when i is completed and completed$(i) = \bot$, otherwise. We define c-by and w-for as partial functions $\mathbb{T} \rightharpoonup \mathbb{T}$ with the meaning that c-by$(i) = j$, resp., w-for$(i) = j$, iff i is called by j, resp., i is waiting for j. We set w-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) $[P] = (\mathbb{C}, \mathbb{A}\text{ct}, \text{ps}_0, \to)$ where $\mathbb{C}$ is the set of program configurations, $\mathbb{A}$ct is a set of transition labels called *actions*, ps_0 is the initial configuration, and $\to \subseteq \mathbb{C} \times \mathbb{A}\text{ct} \times \mathbb{C}$ is the transition relation. Each program statement is interpreted as a transition in $[P]$. The set of actions is defined by:

$$\mathbb{A}\text{ct} = \{(i, ev) : i \in \mathbb{T}, ev \in \{\text{rd}(x), \text{wr}(x), \text{call}(j), \text{await}(k), \text{return}, \text{cont} : j \in \mathbb{T}, k \in \mathbb{T} \cup \{*\}, x \in \mathbb{PV}\}\}$$

The transition relation $\to$ is defined in Fig. 8. Transition labels are written on top of $\to$.

Transitions labeled by $(i, \text{rd}(x))$ and $(i, \text{wr}(x))$ 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 $(i, \text{call}(j))$ 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 completed(j) to $\bot$), and c-by is updated to track its caller (c-by$(j) = i$). A transition (i, return) represents the return from task i. Task i is removed from the stack of currently executing tasks, and completed(i) is set to $\top$ to record the fact that task i is finished.

A transition $(i, \text{await}(j))$ corresponds 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., completed$[j] = \top$), 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 w-for is updated to track the waiting-for relationship (w-for$(i) = j$). Similarly, a transition $(i, \text{await}(*))$ corresponds to task i waiting asynchronously for the completion of an unspecified task. Non-deterministically, task i continues

$$\frac{m \in \mathbb{M} \quad \mathsf{r} := \mathsf{x} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \ell' = \ell[r \mapsto \mathsf{g}(x), \mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, *, *, *, *) \xrightarrow{(i, \mathsf{rd}(x))} (\mathsf{g}, (i, m, \ell') \circ \mathsf{stack}, *, *, *, *)}$$

$$\frac{m \in \mathbb{M} \quad \mathsf{x} := \mathsf{le} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))] \quad \mathsf{g}' = \mathsf{g}[x \mapsto \ell(\mathsf{le})]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, *, *, *, *) \xrightarrow{(i, \mathsf{wr}(x))} (\mathsf{g}', (i, m, \ell') \circ \mathsf{stack}, *, *, *, *)}$$

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

$$\frac{m \in \mathbb{M} \wedge \mathsf{return} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \mathsf{completed}' = \mathsf{completed}[i \mapsto \top]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, *, \mathsf{completed}, *, *) \xrightarrow{(i, \mathsf{return})} (\mathsf{g}, \mathsf{stack}, *, \mathsf{completed}', *, *)}$$

$$\frac{m \in \mathbb{M} \quad \mathsf{await}\ \mathsf{r} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \mathsf{completed}(\ell(r)) = \top \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, *, \mathsf{completed}, *, *) \xrightarrow{(i, \mathsf{await}(\ell(r)))} (\mathsf{g}, (i, m, \ell') \circ \mathsf{stack}, *, \mathsf{completed}, *, *)}$$

$$\frac{m \in \mathbb{M} \quad \mathsf{await}\ \mathsf{r} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \mathsf{completed}(\ell(r)) = \bot \quad \text{w-for}' = \text{w-for}[i \mapsto \ell(r)] \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pending}, \mathsf{completed}, *, \text{w-for}) \xrightarrow{(i, \mathsf{await}(\ell(r)))} (\mathsf{g}, \mathsf{stack}, \{(i, m, \ell')\} \uplus \mathsf{pending}, \mathsf{completed}, *, \text{w-for}')}$$

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

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

$$\frac{m \in \mathbb{M} \quad \text{w-for}(i) = j \quad \mathsf{completed}(j) = \top}{(\mathsf{g}, \epsilon, \{(i, m, \ell)\} \uplus \mathsf{pending}, \mathsf{completed}, *, \text{w-for}) \xrightarrow{(i, \mathsf{cont})} (\mathsf{g}, (i, m, \ell), \mathsf{pending}, \mathsf{completed}, *, \text{w-for})}$$

$$\frac{m \in \mathbb{M} \quad \text{w-for}(i) = *}{(\mathsf{g}, \mathsf{stack}, \{(i, m, \ell)\} \uplus \mathsf{pending}, *, *, \text{w-for}) \xrightarrow{(i, \mathsf{cont})} (\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pending}, *, *, \text{w-for})}$$

Fig. 8. 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. We use init to represent the initial state of a method call.

to the next statement, or task i is interrupted and transferred to the set of pending tasks (w-for(i) is set to $*$).

A transition (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., w-for(i) = j), or an unspecified task (i.e., w-for(i) = $*$). In the first case, the transition is enabled only when the call stack is empty and the task j is completed. In the second case, the transition is enabled without any additional requirements. 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 such methods, it is however complete when focusing on the existence of data races as in our approach.

An execution of P is a sequence $\rho = \mathrm{ps}_0 \xrightarrow{a_1} \mathrm{ps}_1 \xrightarrow{a_2} \ldots$ 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 $a_1 \cdot a_2 \cdot \ldots$ 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 $\mathbb{E}\mathrm{x}(P)$.

Table 1. Strict partial orders included in a trace.

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

Traces. The *trace* of an 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 *synchronous 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 an `await` in MO (an invocation can be interrupted only when executing an `await`). $\mathbb{T}\mathsf{r}(P)$ is the set of traces of a program P.

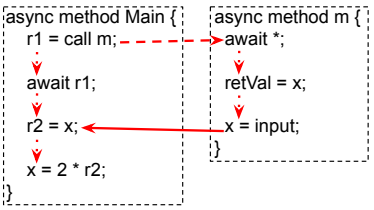

Fig. 9. A trace of an asynchronous program. Arrows between statements denote relations between the corresponding actions in the trace.

Fig. 9 shows a trace where two statements are linked by a dotted arrow if the corresponding actions are related by MO, a dashed arrow if the corresponding actions are related by the CO but not by MO, and a solid arrow if the corresponding actions are related by the HB but not by CO.

4 SYNTHESIZING ASYNCHRONOUS PROGRAMS

We define the synthesis problem we investigate in this work. Given a synchronous program P and a subset of *base* methods $L \subseteq P$, the 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 intended to be models of standard library calls (e.g., IO operations) in a practical context, and asynchronous versions are defined by inserting `await` $*$ statements (in the original synchronous code). We use $P[L]$ to emphasize a subset of base methods L of a program P. Also, we will call L a *library*. A library is called (a)synchronous when all methods are (a)synchronous.

4.1 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 `async`). We assume that each method in L_a corresponds to a method in L with the same name, and vice-versa. A program $P_a[L_a]$ is called an *asynchronization* of

```
method Main {
  r1 = call m;

  r2 = x;

}
method m() {

  retVal = x;
  x = input;
  return;
}
```

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

```
async method Main {
  r1 = call m;

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

Fig. 10. A program and its asynchronizations.

$P[L]$ 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 that satisfies the following: if a method m calls a method $m \in L^*$, then $m \in L^*$. Then, $P_a[L_a]$ is an *asynchronization* of $P[L]$ with respect to L_a if it is obtained from P as follows:

- All methods in $L^* \setminus L$ are declared as asynchronous (we assume that every call to an asynchronous method is followed by an `await` statement, and any method that uses `await` must be declared as asynchronous).
- For each invocation $r :=$ call m of a method $m \in L^*$, add `await` statements await r satisfying the well-formedness syntactic constraints described in Section 3.

For instance, Fig. 10 lists a synchronous program and its two asynchronizations, where $L = L^* = \{m\}$. Asynchronizations differ only in the await placement.

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

4.2 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 $(a_2, a_1) \notin \mathsf{HB}$.

Definition 4.1 (Data Race). 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 $\tau \in \mathbb{T}\mathsf{r}(P_a)$, 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.

For example, the program on the right of Fig. 10 admits a data race between the actions that correspond to `x = input` and `r2 = x`, respectively, in a trace where the call to m is suspended when it reaches await $*$ and the control is transferred to `Main` which executes `r2 = x`. Traces of *synchronous* programs can *not* contain concurrent actions, and therefore they do not admit data races. Note that also $\mathsf{strongAsync}[P, L, L_a]$ does not admit data races.

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

Lemma 4.2. $P[L] \equiv P_a[L_a]$ *implies* $\mathbb{C}[P[L]] = \mathbb{C}[P_a[L_a]]$, *for every* $P_a[L_a] \in \mathsf{Async}[P, L, L_a]$

Proof. We have to show that for any asynchronization P_a of a program P, if P_a does not admit data races then $\mathbb{C}[P_a] = \mathbb{C}[P]$. Let ρ be an execution of P_a that reaches a configuration $\mathsf{ps} \in \mathbb{C}[P_a]$. We show that actions in ρ can be reordered such that any action that occurs in ρ between $(i, \mathsf{call}(j))$ and (j, return) is not of the form $(i, \_)$ (i.e., the task j is executed synchronously). If an action $(i, \_)$ occurs in ρ between $(i, \mathsf{call}(j))$ and (j, return), then it must be concurrent with (j, return). Since P_a does not admit data races, an execution ρ' resulting from ρ by reordering any two concurrent actions reaches the same configuration ps as ρ. Therefore, there exists an execution ρ'' where the actions that occur between any $(i, \mathsf{call}(j))$ and (j, return) are not of the form $(i, \_)$. This is also an execution of P (modulo removing the awaits which have no effect), which implies $\mathsf{ps} \in \mathbb{C}[P]$. □

Definition 4.3. Given a synchronous program $P[L]$, and an asynchronous library L_a, the *asynchronization synthesis problem* asks to enumerate all sound asynchronizations in $\mathsf{Async}[P, L, L_a]$.

There are other approaches for checking behavioral equivalence, e.g., reachable states equivalence, and input-output equivalence, however, they are either undecidable or decidable but highly complex. Our approach offers a stronger notion of equivalence that is simpler to check.

5 ENUMERATING SOUND ASYNCHRONIZATIONS

We describe an algorithm for solving the asynchronization synthesis problem. This algorithm relies on a partial order between asynchronizations that guides the enumeration of possible solutions. It computes optimal solutions (according to this order) repeatedly under different bounds, until exploring the whole space.

5.1 Optimal Asynchronization

We define a partial order on the space of asynchronizations which takes into account the distance between call statements and corresponding await statements.

An `await` statement s_w in a method m of an asynchronization $P_a[L_a] \in \mathsf{Async}[P, L, L_a]$ *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 original synchronous program $P[L]$. Since asynchronizations are obtained by adding `await`s, every call statement in an asynchronization $P_a[L_a] \in \mathsf{Async}[P, L, L_a]$ corresponds to a *fixed* call in $P[L]$.

Definition 5.1. For two asynchronizations $P_a, P'_a \in \mathsf{Async}[P, L, L_a]$, P_a is *less asynchronous* than P'_a, denoted by $P_a \leq P'_a$, iff for every `await` statement s_w in P_a, there exists an `await` statement 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. 10 are ordered by $\leq$ since $\mathsf{Cover}(\texttt{await } r1) = \{\}$ in the first and $\mathsf{Cover}(\texttt{await } r1) = \{\texttt{r2 = x}\}$ in the second.

Note that the strong asynchronization is less asynchronous than any other asynchronization. Also, note that $\leq$ has a unique maximal element that is called the weakest asynchronization and denoted by $\mathsf{weakAsync}[P, L, L_a]$. For instance, the program on the right of Fig. 10 is the weakest asynchronization of the synchronous program on the left of the figure.

Relative Optimality. 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 (w.r.t. $\leq$) and that is sound. Formally, given $P_a \in \mathsf{Async}[P, L, L_a]$, an asynchronization P'_a is called an *optimal asynchronization of P relative to P_a* if (1) $P'_a \leq P_a$, P'_a is sound, and (2) P'_a is maximal among other sound asynchronizations smaller than P_a, i.e., $\forall\, P''_a \in \mathsf{Async}[P, L, L_a]$. P''_a is sound and $P''_a \leq P_1 \Rightarrow P''_a \leq P'_a$.

The following lemma shows that for a given P_a there exists a unique P'_a that is an optimal asynchronization of P relative to P_a. The existence is implied by the fact that $\mathsf{strongAsync}[P, L, L_a]$ is the bottom element of $\leq$. To prove uniqueness, we assume by contradiction that there exist two incomparable optimal asynchronizations P^1_a and P^2_a and select the first await statement s^1_w, according to 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 an optimal asynchronization.

Lemma 5.2. *Given an asynchronization $P_a \in \mathsf{Async}[P, L, L_a]$, there exists a unique program P'_a that is an optimal asynchronization of P relative to P_a.*

Proof. Since $\mathsf{strongAsync}[P, L, L_a]$ is the bottom element of $\leq$, then there always exists a sound asynchronization smaller than P_a. Assume by contradiction that there exist two distinct programs

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

```
1: procedure AsyncSynthesis(P_a, s_w)
2:     P'_a ← OptRelative(P_a);
3:     output P'_a;
4:     P ← ImmPred(P'_a, s_w);
5:     for each (P''_a, s''_w) ∈ P
6:         AsyncSynthesis(P''_a, s''_w);
```

P_a^1 and P_a^2 that are both optimal asynchronizations of P relative to P_a. Let ρ^1 (resp., ρ^2) be an execution of P_a^1 (resp., P_a^2) where every await $*$ does not suspend the execution of the current task, i.e., ρ^1 and ρ^2 simulate the synchronous execution of P. Let s_w^1 be the statement corresponding to the first `await` action in ρ^1 such that (1) there exists an `await` action in ρ^2 with the corresponding `await` statement s_w^2, such that s_w^1 and s_w^2 match the same call in P, and $\mathsf{Cover}(s_w^1) \subset \mathsf{Cover}(s_w^2)$ (this holds because P_a^1 and P_a^2 are distinct asynchronizations of the same synchronous program, thus $\mathsf{Cover}(s_w^1)$ and $\mathsf{Cover}(s_w^2)$ must be comparable), and (2) for every other `await` statement s_w^3 in P_a^1 that generates an `await` action which occurs before the `await` action of s_w^1 in ρ^1, there exists an `await` statement s_w^4 in P_a^2 matching the same call in P, such that $\mathsf{Cover}(s_w^3) = \mathsf{Cover}(s_w^4)$.

Let P_a^3 be the program obtained from P_a^1 by moving the await s_w^1 down (further away from the matching call) such that $\mathsf{Cover}(s_w^1) = \mathsf{Cover}(s_w^2)$. Moving an await down can only create data races between actions that occur after the execution of the matching call. Then, P_a^3 contains a data race iff there exists an execution ρ of P_a^3 and two concurrent actions a_1 and a_2 that occur between the action $(i, \mathsf{await}(j))$ generated by s_w^1 and the action $(i, \mathsf{call}(j))$ of the call matching s_w^1, such that:

$$((i, \mathsf{call}(j)), a_1) \in \mathsf{CO},\ (a_1, a_w) \notin \mathsf{HB},\ ((i, \mathsf{call}(j)), a_2) \in \mathsf{CO} \text{ and } (a_2, (i, \mathsf{await}(j))) \in \mathsf{HB}$$

where the action a_w corresponds to the first await action in the task j. Let s_w be the statement corresponding to the action a_w. Since the only difference between P_a^3 and P_a^2 is the placement of awaits then $((i, \mathsf{call}(j)), a_1) \in \mathsf{CO}$ and $((i, \mathsf{call}(j)), a_2) \in \mathsf{CO}$ hold in any execution ρ' of P_a^2 that contains the actions a_1 and a_2. Also, note that since a_w occurs in the task j that the action of s_w^1 is waiting for. This implies that in ρ^1 the action of s_w occurs before the action of s_w^1 in ρ^1. Therefore, by the definition of s_w^1 we have that s_w in P_a^1 covers the same set of statements as the corresponding s'_w in P_a^2 that matches the same call as s_w. Consequently, $(a_1, a'_w) \notin \mathsf{HB}$ and $(a_2, (i, \mathsf{await}(j))) \in \mathsf{HB}$ hold in any execution ρ' of P_a^2 that contains the actions a_1 and a_2 (a'_w is the action of s'_w). Thus, there exists an execution ρ' of P_a^2 such that the actions a_1 and a_2 are concurrent. This implies that if P_a^3 admits a data race, then P_a^2 admits a data race between actions generated by the same statements. As P_a^2 is data race free, we get that P_a^3 is data race free as well. Since $P_a^1 < P_a^3$, we get that P_a^1 is not optimal, which contradicts the hypothesis. □

5.2 Enumeration Algorithm

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

First, we ignore the second argument of AsyncSynthesis (written in blue), which represents an `await` instruction. For an asynchronization P_a, AsyncSynthesis outputs *all* sound asynchronizations that are smaller than P_a w.r.t. $\leq$. It uses OptRelative to compute the optimal asynchronization P'_a of P relative to P_a, and then, calls itself recursively for all immediate predecessors of P'_a w.r.t. $\leq$. AsyncSynthesis outputs all sound asynchronizations of P when given as input the weakest asynchronization of P. The delay complexity of this algorithm remains exponential in general, because it may output a sound asynchronization multiple times. Indeed, because asynchronizations

are only partially ordered by $\leq$, different chains of recursive calls starting in different immediate predecessors may end up outputting the same asynchronization. For instance, looking at the asynchronizations of our motivating example on the right of Fig. 4, the asynchronization $(0,0,0,0)$ will be outputted twice because it is an immediate predecessor of both $(0,1,0,0)$ and $(1,0,0,0)$.

To avoid outputting the same solution twice, we use a refinement of the above that *restricts* the set of immediate predecessors available for a (recursive) call of AsyncSynthesis. 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, AsyncSynthesis 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 an optimal 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 ImmPred that returns pairs of immediate predecessors and await statements defined as follows:

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

($P'_a \uparrow s''_w$ 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 $(1,1,0,0)$ on the right of Fig. 4, $(0,1,0,0)$ is obtained by moving the *first* await in $\prec_w$ and therefore, after computing the optimal solution relative to it, which is itself, it will explore no more immediate predecessors (ImmPred returns $\emptyset$ because the input s_w is the minimal element of $\prec_w$, and it is already immediately after the matching call). Its immediate predecessor will be explored when recursing on $(1,0,0,0)$. The complexity analysis also relies on a property of the optimal asynchronization relative to an immediate predecessor: if the predecessor is defined by moving an await s''_w, then the optimal asynchronization is obtained by moving only awaits smaller than s''_w w.r.t. $\prec_w$.

Lemma 5.3. *If P''_a is an immediate predecessor of a sound asynchronization P'_a, which is defined by moving an await s''_w in P'_a up, then the optimal sound asynchronization relative to P''_a is obtained by moving only awaits smaller than s''_w w.r.t. $\prec_w$.*

Proof. Moving an await up in P'_a can only create data races between actions that occur after the execution of this await (because the invocation is suspended earlier). The only possible repairs of these data races consists in either moving s''_w down which results in P'_a or moving up some other awaits that occur in methods that (indirectly) call the method in which s''_w occurs. The first case is not applicable because it gives a program that is not smaller than P''_a. In the second case, every await s'_w that is moved up occurs in a method that (indirectly) calls the method in which s''_w occurs, and therefore, s'_w is smaller than s''_w w.r.t. $\prec_w$. □

We show that Algorithm 1 returns all sound asynchronizations when called with the weakest asynchronization and the maximum await in $\prec_w$. Lemma 5.3 shows that after having computed an optimal sound asynchronization P'_a in a recursive call with parameter s_w any smaller sound asynchronization is also smaller than some predecessor in $\mathsf{ImmPred}(P'_a, s_w)$. Thus, the restriction to a subset of predecessors is without loss of completeness (see also the supplementary material).

Theorem 5.4. AsyncSynthesis($\mathsf{weakAsync}[P, L, L_a]$, s_w), *where s_w is maximal in* $\mathsf{weakAsync}[P, L, L_a]$ *w.r.t. $\prec_w$, outputs all sound asynchronizations of $P[L]$ w.r.t. L_a.*

Viewing asychronization synthesis as an enumeration problem, the following theorem states its *delay* complexity in terms of an oracle O_{opt} that returns an optimal asynchronization relative to a given one. This follows from the fact that Algorithm 1 cannot return the same asynchonization

twice and computing immediate predecessors is polynomial time. The former is a consequence of Lemma 5.3. Thus, let P_a^1 and P_a^2 be two predecessors in $\mathsf{ImmPred}(P'_a, s_w)$ obtained by moving up the awaits s_w^1 and s_w^2, respectively, and assume that $s_w^1 \prec_w s_w^2$. By Lemma 5.3, all solutions computed in the recursive call on P_a^1 will have s_w^2 placed as in P'_a while all the solutions computed in the recursive call on P_a^2 will have s_w^2 closer to the matching call. Therefore, the sets of solutions computed in these two branches of the recursion are distinct and the same solution cannot be outputted twice.

Theorem 5.5. *The delay complexity of the asychronization synthesis problem is polynomial time modulo O_{opt}.*

6 COMPUTING OPTIMAL ASYNCHRONIZATIONS

We describe an approach for computing the optimal asynchronization relative to a given synchronization 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.

6.1 Data Race Ordering

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

```
method Main {
  while *
    if *
      r1 = x;
    r2 = y; }
```

Fig. 11

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[2] by two read/write statements s and s', resp., we use $a \prec_{\mathsf{SO}} a'$ to denote the fact that $s \prec s'$ and either $s' \not\prec s$ or s' is reachable from s in the interprocedural[3] control-flow graph of P without taking any back edge[4].

For a *deterministic* synchronous program (admitting a single execution), $a \prec_{\mathsf{SO}} a'$ iff $s \prec s'$. For non-deterministic programs, when s and s' are contained in a loop body, it is possible that $s \prec s'$ and $s' \prec s$. For instance, the statements `r1 = x` and `r2 = y` of the program in Fig. 11 can be executed in different orders depending on the number of loop iterations and whether the if branch is entered during the first loop iteration. In this case, we use the control-flow order to break the tie between a and a'.

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

Fig. 12

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 6.1 (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 $a_1 \prec_{\mathsf{SO}} a_3$.

Example 6.2. For the program in Fig. 12, we have the following order between data races: $(\texttt{x = input}, \texttt{r2 = x}) \prec_{\mathsf{SO}} (\texttt{retVal = x}, \texttt{x = r2 + 1})$ because `r2 = x` is executed before the write

[2]Each action labels a transition in the operational semantics (Section 3), and each transition corresponds to executing a statement. This statement is said to generate the action.

[3]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.

[4]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.

`x = r2 + 1` in the original synchronous program (for simplicity we use statements instead of actions). However, the data races (`x = input`, `r2 = x`) and (`x = input`, `r3 = x`) are incomparable.

6.2 Repairing Data Races

Repairing a data race (a_1, a_2) reduces to modifying the position of a certain `await`. In general, we can either move an `await` down (further away from the matching call), for instance in the method executing a_1, or move an `await` up (closer to the matching call), for instance in the method executing a_2. For example, the data race between `x = input` and `r2 = x` on the right of Fig. 10 can be repaired by either moving `await` $*$ in m after the write `x = input`, so the call to m is suspended later, or `await r1` in `Main` before `r2 = x`, to restrict the set of statements that can execute before m finishes. In the following, we consider only repairs where `await`s are moved up. The "completeness" of this set of repairs follows from the particular order in which we enumerate data races. Intuitively, moving the other `await` down would introduce a data race we have already repaired.

In general, a_1 may not occur in a method m' that is called directly by m, as in Fig. 10, but in another method called by m' or even further down the call tree. Similarly, a_2 may not be part of m, but it may be included in another method called by m after calling m' (but before `await` r), and so on. Next, we describe precisely the transformation that suffices to repair a given data race.

Any two racing actions have a common ancestor in the call order CO which is a call action. This is at least the call action of `main`. The least common ancestor of a_1 and a_2 in CO among call actions is denoted by $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$. Formally, $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$ 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 $(a'_c, a_1) \notin \mathsf{CO}$. For instance, the call action corresponding to `r1 = call m` on the right of Fig. 10 is the *least* common ancestor of the racing actions discussed above. The following lemma (see the supplementary material for a proof) shows that this is the asynchronous call for which the matching `await` must be moved in order to repair a given data race. It also identifies the position where the `await` matching $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$ should be moved in order to repair the data race. Intuitively, this is just before a_2 if a_2 is in the same method as $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$, or more generally, just before the last statement in the same method which precedes a_2 in the call order. On the right of Fig. 10, `await r1` has to be moved before the statement `r2 = x`, which plays the role of a_2.

Lemma 6.3. *Let (a_1, a_2) be a data race in a trace τ of an asynchronization P_a, and $a_c = (i, call(j)) = \mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$. Then, τ contains a unique action $a_w = (i, await(j))$ and a unique action a such that:*

- *$(a, a_w) \in \mathsf{MO}$, and a is the latest action in the method order MO such that $(a_c, a) \in \mathsf{MO}$ and $(a, a_2) \in \mathsf{CO}^*$ (CO^* denotes the reflexive closure of CO).*

Lemma 6.3 identifies a sufficient transformation for repairing a data race (a_1, a_2): moving the await s_w generating the action a_w just before the statement s generating a. This is sufficient because it ensures that every statement that follows $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$[5] in call order will be executed before a and before any statement which succeeds a in call order, including a_2. Note that moving the await a_w anywhere after a will not affect the concurrency between a_1 and a_2.

The pair (s_c, s), where s_c is the call statement generating $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$, is called the *root cause* of the data race (a_1, a_2). Let $\mathsf{RepDRace}(P_a, s_c, s)$ be the maximal asynchronization P'_a smaller than P_a w.r.t. $\preceq$, s.t. no await statement

```
async method Main {
  r1 = call m;
  if *
    r2 = x;
  else
    r3 = y;
  await r1;
}
async method m {
  await *
  retVal = x;
  x = input;
  return;
}
```

```
async method Main {
  r1 = call m;
  if *
    await r1;
    r2 = x;
  else
    r3 = y;
    await r1;
}
async method m {
  await *
  retVal = x;
  x = input;
  return;
}
```

Fig. 13. Examples of asynchronizations.

[5]We abuse the terminology and make no distinction between statements and actions.

Algorithm 2 The procedure OptRelative for computing the optimal asynchronization of P relative to P_a. RootCauseMinDRace(P'_a) returns the root cause of a minimal data race of P'_a w.r.t. $\prec_{SO}$, or $\bot$ if P'_a is data race free.

```
1: procedure OptRelative(P_a)
2:     P'_a ← P_a
3:     root ← RootCauseMinDRace(P'_a)
4:     while root ≠ ⊥
5:         P'_a ← RepDRace(P'_a, root)
6:         root ← RootCauseMinDRace(P'_a)
7:     return P'_a
```

matching s_c occurs after s on a CFG path. When the control-flow graph of the method contains branches, the construction of RepDRace(P_a, s_c, s) consists of (1) replacing all `await` statements matching s_c that are reachable in the CFG from s with a single `await` statement placed just before s, and (2) adding additional `await` statements in branches that "conflict" with the branch containing s. This is to ensure the syntactic constraints described in §3. These additional `await` statements are at maximal distance from the corresponding call statement because of the maximality requirement. For instance, to repair the data race between `r2 = x` and `x = input` in the program on the left of Fig. 13, the statement `await r1` must be moved before `r2 = x` in the `if` branch, which implies that another await must be added on the `else` branch. The result is given on the right of Fig. 13.

The following shows that repairing a minimal data race cannot introduce smaller data races (w.r.t. $\prec_{SO}$), which ensures some form of monotonicity when repairing minimal data races iteratively.

Lemma 6.4. *Let P_a be an asynchronization, (a_1, a_2) a data race in P_a that is minimal w.r.t. $\prec_{SO}$, and (s_c, s) the root cause of (a_1, a_2). Then, RepDRace(P_a, s_c, s) does not admit a data race that is smaller than (a_1, a_2) w.r.t. $\prec_{SO}$.*

Proof. The only modification in the program P'_a = RepDRace(P_a, s_c, s) compared to P_a is the movement of the await s_w matching the call s_c to be before the statement s in a method m. The concurrency added in P'_a that was not possible in P_a is between actions (a', a'') generated by statements s' and s'', respectively, as shown in Fig. 14. W.l.o.g., we assume that $(a', a'') \in$ SO. The statements s_1 and s_2 are those generating a_1 and a_2, respectively. The statement s' is related by CO* to some statement in m that follows s, and s'' is related by CO* to some statement that follows the call to m in the caller of m. Note that s' is ordered by $\prec$ after s_2. Since $(a_1, a_2) \in$ SO and $(a', a'') \in$ SO then $s_2 \prec s''$ and $s_1 \prec s'$. Thus, any new data race (a', a'') in P'_a that was not reachable in P_a is bigger than (a_1, a_2). □

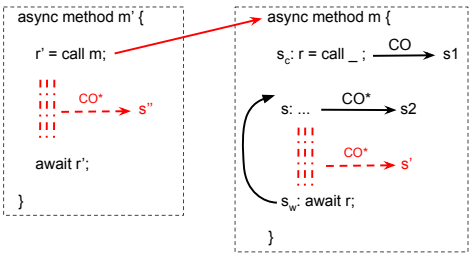

Fig. 14. An excerpt of an asynchronous program.

6.3 A Procedure for Computing Optimal Asynchronizations

Given an asynchronization P_a, the procedure OptRelative in Algorithm 2 computes the optimal asynchronization relative to P_a by repairing data races iteratively until the program becomes data race free. The following theorem states that correctness of this procedure.

Theorem 6.5. *Given an asynchronization $P_a \in$ Async$[P, L, L_a]$, OptRelative(P_a) returns the optimal asynchronization of P relative to P_a.*

Proof. We need to show that any immediate successor P^1_a of the output P'_a = OptRelative(P_a) that is also smaller than P_a (w.r.t. $\leq$) admits data races. By the definition of $\leq$, P^1_a is obtained by moving exactly one await statement s_w in a method m of P'_a further away from the matching call s_c.

Since $P_a^1 \leq P_a$, the position of s_w in the output P'_a is due to repairing a data race between two actions a_1 and a_2 with a root-cause (s_c, s), for some s, on some program $P'_a \leq P''_a \leq P_a$. We show that these actions form a data race in P_a^1. These actions are reachable in an execution of P_a^1 because every method m' that is called by m between s_c and s_w (s_c included), or that follows m' in the call-graph of P_a^1 (or P''_a) has exactly the same code as in P''_a, i.e., the placement of the awaits in those methods is the same as in P''_a (call graphs remain identical between different asynchronizations). This is due to the fact that any data race that would lead to moving an await in one of those methods is before (a_1, a_2) in the order $\prec_{\mathrm{SO}}$. Since s_w in P_a^1 is placed after s, we get that a_1 and a_2 are also concurrent in that execution of P_a^1, which concludes the proof. □

OptRelative(P_a) iterates the process of repairing a data race a number of times which is linear in the size of the input. Indeed, each iteration of the loop results in moving an await closer to the matching call and before at least one more statement from the original synchronous program P.

The fact that data races are enumerated in the order defined by $\prec_{\mathrm{SO}}$ guarantees a bound on the number of times an await matching the same call is moved during the execution of OptRelative(P_a). In general, this bound is the number of statements covered by all the awaits matching the call in the input program P_a. Actually, this is a rather coarse bound. A more refined analysis has to take into account the number of branches in the CFGs. For programs without conditionals or loops, every await is moved at most once during the execution of OptRelative(P_a). In the presence of branches, a call to an asynchronous method may match multiple `await` statements (one for each CFG path starting from the call), and the data races that these `await` statements may create may be incomparable w.r.t. $\prec_{\mathrm{SO}}$. Therefore, for a call statement s_c, let $|s_c|$ be the sum of $|\mathrm{Cover}(s_w)|$ for every await s_w matching s_c in P_a.

Lemma 6.6. *For any asynchronization $P_a \in \mathsf{Async}[P, L, L_a]$ and call statement s_c in P_a, the while loop in OptRelative(P_a) does at most $|s_c|$ iterations that result in moving an await matching s_c.*

Proof. We consider first the case without conditionals or loops, and we show by contradiction that every `await` statement s_w is moved at most once during the execution of OptRelative(P_a), i.e., there exists at most one iteration of the while loop which changes the position of s_w. Suppose that the contrary holds for an `await` s_w. Let (a_1, a_2), and (a_3, a_4) be the data races repaired by the first and second moves of s_w, respectively. By Lemma 6.3, there exist two actions a and a' such that

$$(a_c, a) \in \mathrm{MO},\ (a, a_2) \in \mathrm{CO}^*, (a, a_w) \in \mathrm{MO} \text{ and } (a_c, a') \in \mathrm{MO},\ (a', a_4) \in \mathrm{CO}^*, (a', a_w) \in \mathrm{MO}$$

where $a_w = (i, \mathsf{await}(j))$ and $a_c = (i, \mathsf{call}(j))$ are the asynchronous call action and the matching await action. Let s_2 and s_4 be the statements generating the two actions a_2 and a_4, respectively. Then, we have either $s_2 \prec s_4$ or $s_2 = s_4$, and both cases imply that $(a, a') \in \mathrm{MO}^*$. Thus, moving the await statement generating a_w before the statement generating a implies that it is also placed before the statement generating a' (that occurs after a in the same method). Thus, the first move of the await s_w repaired both data races, which is contradiction.

In the presence of conditionals or loops, moving an await up in one branch may correspond to adding multiple awaits in the other conflicting branches. Also, one call in the program may correspond to multiple awaits on different branches. However, every repair of a data race consists in moving one await closer to the matching call s_c and before one more statement covered by some await matching s_c in the input P_a. □

6.4 Computing Root Causes of Minimal Data Races

We present a reduction from the problem of computing root causes of minimal data races to reachability (assertion checking) in sequential programs. This reduction builds on a program instrumentation for checking if there exists a minimal data race that involves two given statements

```
1  Add before s1:
2  if ( lastTaskDelayed == ⊥ && * )
3    lastTaskDelayed := myTaskId();
4    DescendantDidAwait := thisHasDoneAwait;
5    return

7  Add before s2:
8    if ( task_sc == myTaskId() )
9      s := s2;
10   assert (lastTaskDelayed == ⊥ || !DescendantDidAwait);

13 Replace every statement ``await r'' with:
14   if( r == lastTaskDelayed  ) then
15     if ( !DescendantDidAwait )
16       DescendantDidAwait := thisHasDoneAwait;
17     lastTaskDelayed := myTaskId();
18     return
19   else
20     thisHasDoneAwait := true

22 Add before every statement ``r := call m'':
23   if ( task_sc == myTaskId() ) then
24     s := this statement;

26 Add after every statement ``r := call m'':
27   if ( r == lastTaskDelayed )
28     sc := this statement;
29     task_sc := myTaskId();
```

Fig. 15. A program instrumentation for computing the root cause of a minimal data race between the statements s_1 and s_2 (if any). All variables except for `thisHasDoneAwait` are program (global) variables. `thisHasDoneAwait` is a local variable. The value $\bot$ represents an initial value of a variable. The variables s_c and s store the (program counters of the) statements representing the root cause. The method `myTaskId` returns the id of the current task.

(s_1, s_2), whose correctness relies on the assumption that another pair of statements cannot produce a smaller data race. This instrumentation is used in an iterative process where pairs of statements are enumerated according to the colexicographic order induced by $\prec$. This specific enumeration ensures that the assumption made for the correctness of the instrumentation is satisfied.

Given an asynchronization P_a, the instrumentation described in Fig. 15 represents a synchronous program where all `await` statements are replaced with synchronous code (lines 14–20). This instrumentation simulates asynchronous executions of P_a where methods may be only partially executed, modeling `await` interruptions. It reaches an error state (see the assert at line 10) when an action generated by s_1 is concurrent with an action generated by s_2, which represents a data race, provided that s_1 and s_2 access a common program variable (these statements are assumed to be given as input). Also, the values of s_c and s when reaching the assertion violation represent the root-cause of this data race.

The instrumentation simulates an execution of P_a to search for a data race as follows (we discuss the identification of the root-cause afterwards):

- It executes under the synchronous semantics until an instance of s_1 is non-deterministically chosen as a candidate for the first action in the data race (s_1 can execute multiple times if it is included in a loop for instance). The current invocation is interrupted when it is about to execute this instance of s_1 and its task id t_0 is stored into `lastTaskDelayed` (see lines 2–5).
- Every invocation that transitively called t_0 is interrupted when an `await` for an invocation in this call chain (whose task id is stored into `lastTaskDelayed`) would have been executed in the asynchronization P_a (see line 18).
- Every other method invocation is executed until completion as in the synchronous semantics.
- When reaching s_2, if s_1 has already been executed (`lastTaskDelayed` is not $\bot$) and at least one invocation has only partially been executed, which is recorded in the boolean flag `DescendantDidAwait` and which means that s_1 is concurrent with s_2, then the instrumentation stops with an assertion violation.

A subtle point is that the instrumentation may execute code that follows an `await` r even if the task r has been executed only partially, which would not happen in an execution of the original P_a. Here, we rely on the assumption that there exist no data race between that code and the rest of the task r. Such data races would necessarily involve two statements which are before s_2 w.r.t. $\prec$.

Therefore, the instrumentation is correct only if it is applied by enumerating pairs of statements (s_1, s_2) w.r.t. the colexicographic order induced by $\prec$.

Next, we describe the computation of the root-cause, i.e., the updates on the variables s_c and s. By definition, the statement s_c in the root-cause should be a call that makes an invocation that is in the call stack when s_1 is reached. This can be checked using the variable `lastTaskDelayed` that stores the id of the last such invocation popped from the call stack (see the test at line 27). The statement s in the root-cause can be any call statement that has been executed in the same task as s_c (see the test at line 23), or s_2 itself (see line 9).

Let $[[P_a, s_1, s_2]]$ denote the instrumentation in Fig. 15. We say that the values of s_c and s when reaching the assertion violation are the root cause computed by this instrumentation. The following theorem states its correctness.

Theorem 6.7. *If $[[P_a, s_1, s_2]]$ reaches an assertion violation, then it computes the root cause of a minimal data race, or there exists (s_3, s_4) such that $[[P_a, s_3, s_4]]$ reaches an assertion violation and (s_3, s_4) is before (s_1, s_2) in colexicographic order w.r.t. $\prec$.*

Based on Theorem 6.7, we define an implementation of the procedure RootCauseMinDRace(P_a) used in computing optimal asynchronizations (Algorithm 2) as follows:

- For all pairs of read or write statements (s_1, s_2) in colexicographic order w.r.t. $\prec$.
  - If $[[P_a, s_1, s_2]]$ reaches an assertion violation, then
    - return the root cause computed by $[[P_a, s_1, s_2]]$
- return $\bot$

The order $\prec$ between read or write statements can be computed using a quadratic number of reachability queries in the synchronous program P. Therefore, $s \prec s'$ iff an instrumentation of P that sets a flag when executing s and asserts that this flag is not set when executing s' reaches an assertion violation. The following theorem states the correctness of the procedure above.

Theorem 6.8. *RootCauseMinDRace(P_a) returns the root cause of a minimal data race of P_a w.r.t. $\prec_{SO}$, or $\bot$ if P'_a is data race free.*

This procedure performs a quadratic number of reachability queries in sequential programs.

Theorem 6.9. *The complexity of RootCauseMinDRace is polynomial time modulo an oracle for the reachability problem in sequential programs.*

6.5 Asymptotic Complexity of Asynchronization Synthesis

We state the complexity of the asynchronization synthesis problem. Theorem 5.5 shows that its delay complexity is polynomial modulo the complexity of OptRelative in Algorithm 2, which by the results in this section, reduces to a polynomial number of reachability queries in sequential programs. The reachability problem is PSPACE-complete for finite-state sequential programs [Godefroid and Yannakakis 2013].

Theorem 6.10. *The output complexity[6] 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).

Theorem 6.11. *Checking whether there exists a sound asynchronization different from the strong asynchronization is PSPACE-complete.*

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

Proof. For hardness, checking if a sequential program P reaches a particular control location ℓ can be reduced to the non-existence of a non-trivial sound asynchronization of a program P' defined as follows: (1) define a new method m that writes to a new program variable x, and insert a call to m followed by a write to x at location ℓ, and (2) insert a write to x after every call statement that calls a method in $\{m'\}^*$, where m' is the method containing ℓ. Let m_a be an asynchronous version of m obtained by inserting an `await` $*$ at the beginning. Then, ℓ is reachable in P iff the only sound asynchronization of P' w.r.t. $\{m_a\}$ is the strong asynchronization. □

7 OPTIMAL ASYNCHRONIZATIONS USING DATA-FLOW ANALYSIS

We present a procedure for computing sound asynchronizations, based on a bottom-up inter-procedural data-flow analysis. It computes *optimal* asynchronizations for abstractions of programs where every Boolean condition in if-then-else statements or while loops is replaced with the non-deterministic choice $*$.

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 waited for in the same iteration (see the syntactic constraints in §3).

Theorem 7.1. *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.*

We present a procedure for computing optimal asynchronizations of $P^\#$, relative to a given asynchronization $P_a^\#$. This procedure 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 Section 6.2. 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 whose purpose is 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[7]. 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 RW-var$(m) = (\mathcal{R}(m), \mathcal{W}(m))$. We extend the notation RW-var to statements as follows: RW-var$(\langle r \rangle := \langle x \rangle) = (\{x\}, \emptyset)$, RW-var$(\langle x \rangle := \langle le \rangle) = (\emptyset, \{x\})$, RW-var$(r :=$ `call` $m)$ = RW-var(m), and RW-var$(s) = (\emptyset, \emptyset)$, for any other type of statement s. Also, let CRW-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 CRW-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
  - CRW-var$(m) \leftarrow$ CRW-var$(m) \cup$ CRW-var(m')
- If s is reachable from an `await` statement in the CFG of m
  - CRW-var$(m) \leftarrow$ CRW-var$(m) \cup$ RW-var(s)

[7]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.

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 OptRelative$^\#$ 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 $r := \texttt{call } m'$ and $\texttt{await } r$ (for some m'), and RW-var$(s) \bowtie$ CRW-var(m'), then $P_a^\# \leftarrow \mathsf{RepDRace}(P_a^\#, r := \texttt{call } m', s)$
- Return $P_a^\#$

The following theorem states the correctness of OptRelative$^\#$. This procedure repairs data races in an order which is $\prec_{\mathrm{SO}}$ with some exceptions that do not affect optimality, i.e., the number of times an await matching the same call can be moved. For instance, if a method m calls two other methods m_1 and m_2 in this order, the procedure above may handle m_2 before m_1, i.e., repair data races between actions that originate from m_2 before data races that originate from m_1, although the former are bigger than the latter in $\prec_{\mathrm{SO}}$. This does not affect optimality because those repairs are "independent", i.e., any repair in m_2 cannot influence a repair in m_1, and vice-versa. The crucial point is that this procedure repairs data races between actions that originate from a method m before data races that involve actions in methods preceding m in the call graph, which are bigger in $\prec_{\mathrm{SO}}$ than the former.

Theorem 7.2. *OptRelative$^\#(P_a^\#)$ returns an optimal asynchronization relative to $P_a^\#$.*

Since OptRelative$^\#$ is based on a single bottom-up traversal of the call graph of the input asynchronization $P_a^\#$, Theorem 5.5 implies the following result.

Theorem 7.3. *The delay complexity of the asynchronization synthesis problem restricted to abstracted programs $P^\#$ is polynomial time.*

8 MULTI-THREADED REFACTORINGS

We discuss an extension of our framework to *multi-threaded refactorings* that rewrite a sequential program into a multi-threaded program where every method invocation is executed on a different thread. A caller can wait for a callee to complete using a `join` primitive. A `start` primitive for spawning a new thread is the counterpart of an asynchronous call while `join` is the counterpart of `await`. For instance, Fig. 16 lists a sequential program, a possible asynchonization, and a multi-thread refactoring (both refactorings place the awaits/joins as far away as possible from the calls).

An important difference between start/join and async/await is the happens-before order relation. For instance, the asynchronization on the center of Fig. 16 assigns 1 to x (line 11) before it assigns 2 to x (line 4), as in the original sequential program. However, the multi-thread program on the right of Fig. 16 may execute these two assignments in any order, and admits a behavior that is not possible in the sequential program (assigning 2 before assigning 1). Repairing this data-race consists in moving the `join` at line 5 to occur before assigning 2 to x at line 4. In general, the happens-before order is weaker compared to an analogous asynchronization, where awaits are placed as the joins, which implies that any multi-threaded refactoring can be rewritten to an asynchronization. The vice-versa may not be possible as shown in this example.

Despite this difference, it can still be proved that there exists a unique multi-threaded refactoring that is sound, i.e., does not admit data races, and optimal, i.e., maximizes the distance between start and join, a result similar to Lemma 5.2. Assuming by contradiction the existence of two incomparable optimal and sound refactorings, one can show that moving a join in one refactoring further away from the matching call as in the other refactoring does not introduce data races (contradicting optimality). To compute optimal and sound multi-threaded refactorings, one can

```
1 void Main() {
2  F();

4  x = 2;

6 }

8 void F() {
9  IO();

11  x = 1;

13 }
```

```
1 async Task MainAsync() {
2  Task t1 = F();

4  x = 2;
5  await t1;
6 }

8 async Task F() {
9  Task t2 = IOAsync();

11  x = 1;
12  await t2;
13 }
```

```
1 void Main() {
2  Thread thr1 = new Thread(F);
3  thr1.Start();
4  x = 2;
5  thr1.Join();
6 }

8 void F() {
9  Thread thr2 = new Thread(IO);
10  thr2.Start();
11  x = 1;
12  thr2.Join();
13 }
```

Fig. 16. A synchronous C# program, an asynchronization, and a multi-threaded refactoring.

apply the same iterative process of repairing data-races (the happens-before reflects multi-threading instead of async/await), prioritizing data races involving statements that would execute first in the sequential program. The repairing of a data-race is similar and consists in moving a join up.

In contrast to async/await, moving a join up does not introduce new data races (since no new parallelism is introduced). This implies that all the predecessors of a sound multi-threaded refactoring are also sound, i.e., the set of sound multi-threaded refactorings is downward closed.

9 EXPERIMENTAL EVALUATION

We present an empirical evaluation of our asynchronization enumeration approach, where optimal asynchronizations are computed using the data-flow analysis described in Section 7. We consider a benchmark consisting mostly of asynchronous C# programs extracted from open-source GitHub projects. We evaluate the effectiveness of our technique 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 developped a prototype tool that relies on the Roslyn .NET compiler platform [Roslyn 2021] to construct CFGs for methods in a given C# program. This prototype supports C# programs written in SSA form that include basic conditional or looping constructs and async/await as concurrency primitives. It assumes that any alias information is provided apriori; these constraints can be removed in the future with more engineering effort. Object fields are interpreted as program variables in the terminology of the program syntax in Section 3 (data races concern accesses to object fields).

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 outlined in Table 2. This 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 (that have both synchronous and asynchronous versions), which come from 5 C# libraries (*System.IO*, *System.Net*, *Windows.Storage*, *Microsoft.WindowsAzure.Storage*, and *Microsoft.Azure.Devices*). They are modeled as described in Section 3.

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

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 ($await_{\#}$). 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 optimal asynchronization are covering *more than* in the input program (cover), number of computed asynchronizations (async), and running time (t).

Program	loc	m	c	ac	$await_{\#}$	await	races	cover	async	t(s)
SyntheticBenchmark-1	77	3	6	5	4	4	5	0	9	5
SyntheticBenchmark-2	115	4	12	10	6	3	3	0	8	5
SyntheticBenchmark-3	168	6	16	13	9	7	4	0	128	9
SyntheticBenchmark-4	171	6	17	14	10	8	5	0	256	55
SyntheticBenchmark-5	170	6	17	14	10	8	9	0	272	138
Azure-Remote	520	10	14	5	0	0	0	0	1	5
Azure-Webjobs	190	6	14	6	1	1	0	1	3	4
FritzDectCore	141	7	11	8	1	1	0	1	2	5
MultiPlatform	53	2	6	4	2	2	0	2	4	5
NetRpc	887	13	18	11	4	1	3	0	3	5
TestAZureBoards	43	3	3	3	0	0	0	0	1	4
VBForums-Viewer	275	7	10	7	3	2	1	1	6	5
Voat	178	3	6	5	2	1	1	1	4	10
WordpressRESTClient	133	3	10	8	4	2	1	0	4	5
ReadFile-Stackoverflow	47	2	3	3	1	0	1	0	1	6
UI-Stackoverflow	50	3	4	4	3	3	3	0	12	5

asynchronizations depends on the number of invocations (column ac in Table_1) and the size of the code blocks between an invocation and the instruction using its return value (column 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)[8]. This shows that we are able to increase the distances between `await`s and their matching calls for those programs. On average the distance between `await`s and their matching calls in optimal asynchronizations for non synthetic benchmarks is 1.27 statements.

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. Also, this leaves the possibility of making a choice based on other criteria, e.g., performance metrics. While asynchronizations are computed statically, their performance can be derived only dynamically (executing them). In general, we are not aware of any syntactic criteria that can guide towards computing a best solution w.r.t. performance in practice. 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 artifacts are available in an anonymous GitHub repository [Experiments 2021].

10 RELATED WORK

There are many works on synthesizing or repairing concurrent programs in the standard multi-threading model, e.g., automatic parallelization in compilers [Bacon et al. 1994; Blume et al. 1996; Han and Tseng 2001], or synchronization synthesis [Bloem et al. 2014; Cerný et al. 2015, 2013, 2014; Clarke and Emerson 2008; Gupta et al. 2015; Manna and Wolper 1984; Vechev et al. 2009, 2010]. Our paper focuses on the use of the async/await primitives which poses specific challenges that are not covered in these works. For instance, synthesizing lock placements does not admit unique optimal solutions w.r.t. a syntactic order as for async/await.

[8]The synthetic examples are weakest asynchronizations to start with.

Program Refactoring. A number of program refactoring tools have been proposed for converting C# programs using explicit callbacks into async/await programs [Okur et al. 2014], Android programs using AsyncTask into programs that use IntentService [Lin et al. 2015], or sequential applications into parallel applications using concurrent libraries for Java [Dig et al. 2009]. The C# related tool [Okur et al. 2014], which is the closest to our work, makes it possible to repair misusage of async/await that might result in deadlocks. Their repairing mechanism is based on forcing the continuations after the first `await` to run on background threads. This tool cannot modify procedure calls to be asynchronous as in our work. Compared to all these works, we give an algorithmic framework with precise specifications and complexity analysis.

Data Race Detection. There are many works that study dynamic data race detection using happens-before and lock-set analysis, or timing-based detection, e.g., [Flanagan and Freund 2009; Kini et al. 2017; Li et al. 2019; Raman et al. 2010; Smaragdakis et al. 2012]. [Raman et al. 2010] proposes a dynamic data race detector for async-finish task-parallel programs by adapting the algorithm proposed in [Feng and Leiserson 1997] that computes abstract summaries of parallel tasks. [Li et al. 2019] presents a testing technique for finding data races in C# and F# programs, based on inserting timing delays in unsafe methods (e.g., methods that access memory without locking), and a monitor for finding data races. These methods could be used to approximate our reduction from data race checking to reachability in sequential programs.

A number of works [Blackshear et al. 2018; Engler and Ashcraft 2003; Liu and Huang 2018] propose static analyses for finding data races. [Blackshear et al. 2018] 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 [Santhiar and Kanade 2017], they precompute method summaries in order to extract potential racy accesses. These approaches are similar to the analysis we present in Section 7, but they concern a different programming model.

Analyzing Asynchronous Programs. There exist several works that propose program analyses for various classes of asynchronous programs. [Bouajjani and Emmi 2012; Ganty and Majumdar 2012] give complexity results for the reachability problem, and [Santhiar and Kanade 2017] proposes a static analysis for deadlock detection in C# programs that use both asynchronous and synchronous wait primitives. This work relies on the static analysis introduced in [Madhavan et al. 2012] for computing method summaries in terms of points-to relations. [Bouajjani et al. 2017] 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.

11 CONCLUSION

We have proposed a framework for refactoring sequential programs to equivalent asynchronous programs that rely on the async/await primitives. We have determined precise complexity bounds for the problem of computing a sound asynchronization that maximizes the distance between asyncs and awaits, which in theory, increases the level of parallelism, and the problem of computing all sound asynchronizations. The latter problem is useful in a context where performance measures cannot be derived statically, which is usually the case, and makes it possible to compute a sound asynchronization that maximizes performance by separating concerns (enumerate sound asynchronizations and evaluate performance separately). We have also investigated the related problem of synthesizing sound multi-threaded refactorings where every method call is executed by a different thread, showing that our techniques extend quite easily, which witnesses the "robustness" of our framework. 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 extracted from open-source repositories.

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. Our paper deals with these two paradigms separately, but extending our approach for such programs is an interesting direction for future work.

REFERENCES

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (1994), 345–420. https://doi.org/10.1145/197405.197406

Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause 'n' Play: Formalizing Asynchronous C#. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer, 233–257. https://doi.org/10.1007/978-3-642-31057-7_12

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 144:1–144:28. https://doi.org/10.1145/3276514

Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. 2014. Synthesis of synchronization using uninterpreted functions. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 35–42. https://doi.org/10.1109/FMCAD.2014.6987593

William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David A. Padua, Yunheung Paek, William M. Pottenger, Lawrence Rauchwerger, and Peng Tu. 1996. Parallel Programming with Polaris. *IEEE Computer* 29, 12 (1996), 87–81. https://doi.org/10.1109/2.546612

Ahmed Bouajjani and Michael Emmi. 2012. Analysis of recursively parallel programs. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 203–214. https://doi.org/10.1145/2103656.2103681

Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. 2017. Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Hongseok Yang (Ed.), Vol. 10201. Springer, 170–200. https://doi.org/10.1007/978-3-662-54434-1_7

Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2015. From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 180–197. https://doi.org/10.1007/978-3-319-21668-3_11

Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2013. Efficient Synthesis for Concurrency by Semantics-Preserving Transformations. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 951–967. https://doi.org/10.1007/978-3-642-39799-8_68

Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2014. Regression-Free Synthesis for Concurrency. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 568–584. https://doi.org/10.1007/978-3-319-08867-9_38

Edmund M. Clarke and E. Allen Emerson. 2008. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *25 Years of Model Checking - History, Achievements, Perspectives (Lecture Notes in Computer Science)*, Orna Grumberg and Helmut Veith (Eds.), Vol. 5000. Springer, 196–215. https://doi.org/10.1007/978-3-540-69850-0_12

Danny Dig, John Marrero, and Michael D. Ernst. 2009. Refactoring sequential Java code for concurrency via concurrent libraries. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 397–407. https://doi.org/10.1109/ICSE.2009.5070539

Dawson R. Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, Michael L. Scott and Larry L. Peterson (Eds.). ACM, 237–252. https://doi.org/10.1145/945445.945468

Experiments. 2021. https://github.com/asynchronizations/artifact

Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '97, Newport, RI, USA, June 23-25, 1997*, Charles E. Leiserson and David E. Culler (Eds.). ACM, 1–11. https://doi.org/10.1145/258492.258493

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland,*

June 15-21, 2009, Michael Hind and Amer Diwan (Eds.). ACM, 121–133. https://doi.org/10.1145/1542476.1542490

Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 6:1–6:48. https://doi.org/10.1145/2160910.2160915

Patrice Godefroid and Mihalis Yannakakis. 2013. Analysis of Boolean Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science)*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. Springer, 214–229. https://doi.org/10.1007/978-3-642-36742-7_16

Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. 2015. Succinct Representation of Concurrent Trace Sets. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 433–444. https://doi.org/10.1145/2676726.2677008

Hwansoo Han and Chau-Wen Tseng. 2001. A Comparison of Parallelization Techniques for Irregular Reductions. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, USA, April 23-27, 2001*. IEEE Computer Society, 27. https://doi.org/10.1109/IPDPS.2001.924963

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 157–170. https://doi.org/10.1145/3062341.3062374

Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 162–180. https://doi.org/10.1145/3341301.3359638

Yu Lin, Semih Okur, and Danny Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 224–235. https://doi.org/10.1109/ASE.2015.50

Bozhen Liu and Jeff Huang. 2018. D4: fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 359–373. https://doi.org/10.1145/3192366.3192390

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2012. Modular Heap Analysis for Higher-Order Programs. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings (Lecture Notes in Computer Science)*, Antoine Miné and David Schmidt (Eds.), Vol. 7460. Springer, 370–387. https://doi.org/10.1007/978-3-642-33125-1_25

Zohar Manna and Pierre Wolper. 1984. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 6, 1 (1984), 68–93. https://doi.org/10.1145/357233.357237

Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. 2014. A study and toolkit for asynchronous programming in c#. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 1117–1127. https://doi.org/10.1145/2568225.2568309

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings (Lecture Notes in Computer Science)*, Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.), Vol. 6418. Springer, 368–383. https://doi.org/10.1007/978-3-642-16612-9_28

Roslyn. 2021. https://github.com/dotnet/roslyn

Anirudh Santhiar and Aditya Kanade. 2017. Static deadlock detection for asynchronous C# programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 292–305. https://doi.org/10.1145/3062341.3062361

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400. https://doi.org/10.1145/2103656.2103702

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2009. Inferring Synchronization under Limited Observability. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Kowalewski and Anna Philippou (Eds.), Vol. 5505. Springer, 139–154. https://doi.org/10.1007/978-3-642-00768-2_13

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January*

17-23, 2010, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 327–338. https://doi.org/10.1145/1706299.1706338