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 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. Asynchronous programming is essential for programming efficient device drivers, UI-driven, web and cloud applications, etc.

As an alternative to the tedious model of asynchronous programming that required developers to explicitly register callbacks with the asynchronous procedure calls, C# 5.0 [Bierman et al. 2012] introduced the async/await primitives. These primitives allow the developer to write code in the familiar sequential style without explicit callbacks. An asynchronous procedure, declared with the keyword `async`, returns a task object that the caller can use 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 following the await instruction is the continuation that is automatically called back when the result of the callee is ready. Due to its simplicity, this paradigm has become popular across many programming languages, e.g., C++, JavaScript, F#, Python, etc.

While simplifying the writing of asynchronous programs, the async/await primitives introduce concurrency which is notoriously hard to deal with. Depending on the scheduler, the code in between a call and a matching await (matching means that they refer to the same task) may execute before some portion of the awaited task (if the latter passed the control to its caller before finishing), or after the awaited task finished. The resemblance with sequential code can be especially deceitful since this non-determinism is opaque. It is quite common that await instructions are placed immediately after the corresponding asynchronous call which limits the benefits that one can obtain from executing code in the caller concurrently with code in the callee.

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/

In this paper, we address the problem of writing efficient asynchronous code that uses the 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 a hard problem, 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, replacing calls to `StreamReader.ReadToEnd()` in C# for reading the content of a stream with calls to its asynchronous version `StreamReader.ReadToEndAsync()`).

Moreover, we consider an equivalence relation which essentially corresponds to absence of data races in the asynchronous program. As usual, a data race is a pair of conflicting accesses to the same memory location that are not ordered by the control-flow. Relying on absence of data races instead of a more precise equivalence relation like equality of reachable sets of states could in principle prevent enumerating some number of valid asynchronizations. However, checking equality of reachable sets of states is known to be a hard problem in general, and relying on absence of data races is a well established compromise, that is supported as well by our empirical evaluation.

We consider the problem of enumerating *all* data-race free asynchronizations of a sequential program P with respect to substituting a set of methods in P with asynchronous versions. Enumerating all data-race free 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 await statements that do not introduce data races. An important challenge is that *every* method that precedes a substituted one in the call graph must be defined as asynchronous, and therefore, any call to such a method must be followed by an await (the use of async/await imposes a syntactic requirement that every method that includes an await in its body must be declared as asynchronous). Therefore, deriving a data-race free asynchronization may require finding suitable await placements for calls to many more methods than the substituted ones.

In general, the number of (data-race free) 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 outputs. 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.

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 of queries to reachability). 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 the 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 however that this is not necessary because of the control-flow imposed by awaits, i.e., passing the control from callee to caller.

As a more pragmatic approach to these problems, we define a procedure for computing data-race free asynchronizations which relies on a bottom-up interprocedural data-flow analysis. Intuitively,

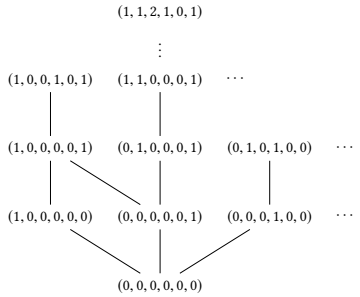
```
1  void Main() {
2      string url1 = ReadFile("file1.txt");
3      string url2 = ReadFile("file2.txt");
6      int val1 = ContentLength(url1);
10     int val2 = ContentLength(url2);
13     int r = x;
15     Debug.Assert(r == val1 + val2);
16 }
18 string ReadFile(string file) {
19     using StreamReader reader = new
         StreamReader(file);
20     string content = reader.ReadToEnd();
22     return content;
23 }
25 int ContentLength(string url) {
26     using System.Net.HttpClient client = new
         HttpClient()
27     string urlContents = client.GetString(url);
28     int r1 = x;
30     x = r1 + urlContents.Length;
31     return urlContents.Length;
32 }
```

```
1  async Task MainAsync() {
2      Task<string> t1 = ReadFile("file1.txt");
3      Task<string> t2 =  ReadFile("file2.txt");
5      string url1 = await t1;
6      Task<int> t3 = ContentLength(url1);
7      int val1 = await t3;
9      string url2 = await t2;
10     Task<int> t4 = ContentLength(url2);
11     int val2 = await t4;
13     int r = x;
15     Debug.Assert(r == val1 + val2);
16 }
18 async Task<string> ReadFile(string file) {
19     using StreamReader reader = new
         StreamReader(file);
20     Task<string> t5 = reader.ReadToEndAsync();
21     string content = await t5;
22     return content;
23 }
25 async Task<int> ContentLength(string url) {
26     using System.Net.HttpClient client = new
         HttpClient();
27     Task<string> t6 =
         client.GetStringAsync(url);
28     int r1 = x;
29     urlContents = await t6;
30     x = r1 + urlContents.Length;
31     return urlContents.Length;
32 }
```

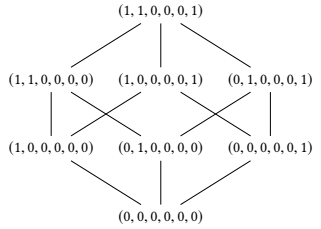
Fig. 1. A sequential (synchronous) C# program and an equivalent asynchronous refactoring (x is a static variable). Every call to an asynchronous method returns a Task object which is given as parameter to awaits.

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 data-race free asynchronizations of abstracted programs.

We implemented the asynchronization enumeration approach based on data-flow analysis in a prototype tool that applies to C# programs. We have evaluated this implementation on a number of non-trivial programs extracted from open source repositories. This evaluation shows that data-race free asynchronizations can be enumerated efficiently and in some cases, we found asynchronizations that increase the amount of parallelism (increasing 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 the async/await primitives.



(a) Space of all asynchronizations.



(b) Space of all data race free asynchronizations.

Fig. 2. Partially-ordered sets of asynchronizations of the program on the left of Fig. 1. The edges connect comparable elements, smaller elements being below bigger elements.

2 OVERVIEW

We demonstrate our synthesis framework on the C# program given on the left of Fig. 1 (we omit syntactic details like class declarations). This is a sequential program with a `Main` method that invokes two other methods `ReadFile` and `ContentLength` in a synchronous way (using the standard call stack semantics). The method `ReadFile` reads and returns the content of a file while `ContentLength` returns the length of the text in a webpage. The URLs given as input to `ContentLength` are read from some files 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 15.

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 in an asynchronous manner can lead to significant improvements in performance. For instance, they could allow time-consuming computations following these calls to proceed and be executed without waiting for the completion of these tasks.

Our work presents an automated approach for synthesizing *equivalent* refactorings of this program where the calls to `StreamReader.ReadToEnd` and `HttpClient.GetString` are replaced with asynchronous counterparts (we assume that the asynchronous versions have the same effect). An example of such a rewriting is given on the right of Fig. 1. In general, 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 20 is immediately followed by an `await` since the next instruction (the `return` at line 22) uses the value computed by `ReadToEndAsync`. Then, a function like `ReadFile` containing an `await` must be declared to be asynchronous itself, which implies that its invocations must also be accompanied

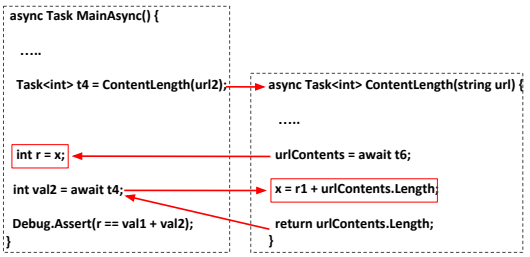


Fig. 3. An execution with a race on x.

[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 in a synchronous way should pass through creating `WebRequest` and `HttpWebResponse` objects. The explanations would be valid in such a case as well.

by suitable `await`s (see the method `MainAsync`). In general, synthesizing an equivalent refactoring like the one on the right of Fig. 1 boils down to finding a correct placement of `await`s for every method call that transitively calls a substituted method (we not consider deeper refactoring techniques like introducing/modifying conditionals or rewriting loops).

The program on the right of Fig. 1 is not the only solution to our synthesis problem. For instance, the `await` at line 29 can be moved one statement up (before the read of `x`) and the resulting program remains equivalent to the original sequential program (because the execution of `GetStringAsync` at line 27 does not access `x`). However, moving the `await` at line 11 after the read of `x` at line 13 leads to new behaviors, not possible in the original program. An example is pictured in Fig. 3 (edges represent execution order). In this execution, the read of `x` in `MainAsync` occurs before the write to `x` in `ContentLength` which leads to an assertion violation. These two statements were executed in the opposite order in the original program. They represent a data race in this asynchronous program because they are not ordered by the control-flow. There exists another execution where they execute in the same order as in the original program: if task `t6` finishes before `ContentLength` does `await t6`, then the `await` has no effect, and `ContentLength` executes until completion before passing the control to its caller `MainAsync`. The notion of equivalence we consider between asynchronous refactorings and sequential programs is defined as absence of data races.

In general, the number of possible asynchronizations is exponential in the size of the program (the number of procedure calls). Asynchronizations can be partially ordered depending on the distance, i.e., the number of statements from the original sequential program, between an await and a matching call. Fig. 2a pictures an excerpt of this partial order where every asynchronization is represented as a vector of distances, the first element is the number of statements between the call and the await on `t1`, and so on. The asynchronization on the right of Fig. 1 corresponds to the vector $(1, 1, 0, 0, 0, 1)$ since for instance, the await on `t1` is separated from the matching call by a single statement from the original program, i.e., the call `ReadFile("file2.txt")`. An asynchronization is smaller than another one (less asynchronous) if their vectors of distances are related by the component-wise extension of the natural number ordering.

The bottom of this partial order, $(0, 0, 0, 0, 0, 0)$, represents an asynchronization where every call is immediately followed by an await. This program has precisely the same semantics as the original sequential program (every call is waited for until it finishes), and it is data race free. The top element is the least asynchronization where the awaits cannot be moved further away from their matching calls because of the use of the return values. For instance, the await on `t1` cannot be moved after the call `ContentLength(url1)` because the return value of `t1` is used as an input. Fig. 2b gives the set of all data race free 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.

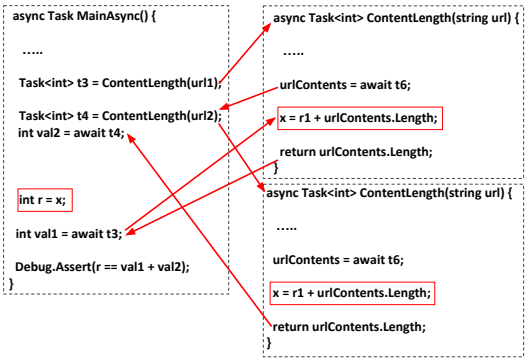


Fig. 4. An execution with two races on x.

To enumerate all data race free asynchronizations, we perform a top-down traversal of the partial order in Fig. 2a. We first compute the biggest element which is data race free. Although this is a partial order, we show that this element is actually *unique*. Then, for each of its immediate successors P_a, we compute the biggest data race free asynchronization which is smaller than P_a

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

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

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

Fig. 5. Program syntax. $\langle m \rangle$, $\langle x \rangle$, and $\langle r \rangle$ represent a method name, a program variable, and a local variable, respectively. $\langle le \rangle$ denotes an expression over local variables, or $*$ which represents non-deterministic choice.

(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. The enumeration finishes when reaching the bottom, which is data race free by definition, on all branches of the recursion.

Computing an optimal asynchronization relative to a given P_a is an iterative process that repairs data races. For instance, the race in Fig. 3 can be repaired by moving the `await t4` one position up, before the read of `x` (this way, the write to `x` will execute before the read). The call to `ContentLength` that matches this await and the read of `x` are regarded as the *root cause* of this data race.

For efficiency, the data races to be repaired are enumerated 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, Fig. 4 pictures an execution of an asynchronization obtained from the one in Fig. 1 by moving the `await t3` at line 7 after the read of `x` at line 13 (we duplicated the code of `ContentLength` to distinguish between the two calls). This execution contains two data races: a data race between the two writes to `x` in the two calls to `ContentLength`, and a data race between the write to `x` in the first call to `ContentLength` and the read of `x` in `MainAsync`. Since the two writes to `x` would occur first in an execution of the original sequential program (before the read of `x` in the other data race), we repair this data race first, by moving the `await t3` before the second call to `ContentLength` (these two calls represent the root cause of the data race). Interestingly, this repair step removes the write-read data race as well. Note that if we would have had repaired these data races in the opposite order, we would have had moved `await t3` before the read of `x`, and then, in a second repair step, before the first call to `ContentLength`.

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. Moreover, 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

We present a simple programming language the includes the async/await primitives, which is used to formalize our approach.

3.1 Syntax

Fig. 5 gives the syntax of our programming language. A *program* is defined by set of methods, including a distinguished `main` method, which are classified as *synchronous* or *asynchronous*. Synchronous methods execute immediately as they are invoked and run continuously until completion. Asynchronous methods, distinguished using the keyword `async`, can run only partially and be interrupted when executing an `await` statement. Following standard syntactic assumptions, 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,...). For simplicity, we abstract away from input and return parameters and assume that they are modeled using dedicated program variables. We assume that each method call returns a *unique task identifier* from a set $\mathbb{T}$ which is mainly used to record control dependencies imposed by `await` statements (for uniformity, we assume that synchronous methods return a task identifier as well). Each other statement is either an assignment to a local variable from the set $\mathbb{LV}$ or to a program variable from the set $\mathbb{PV}$, an `await` statement, a `return` statement, a `while` loop, or an `if-else` conditional statement. We assume that variables take values from an unspecified 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 $\langle x \rangle := \langle le \rangle$, where le is an expression over local variables, 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 have the distinguishing feature of using `await` statements which allows them to wait for the completion of a task (invocation) while the control is passed to their caller. The `await` statement takes a parameter r which species 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 is not modeled explicitly in our programming language, 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 of two IO methods `ReadToEndAsync` and `GetStringAsync` from C# in our programming language. We use a set of variables to represent read or write accesses to system resources such as the network or the file system. These variables are important to model potentially racing accesses to system resources in asynchronous executions. The await for the completion of those accesses is modeled by the `await` $*$. `GetStringAsync` includes a read access to the resource `WWW` (for world wide web) specified by the input url. As mentioned above, method parameters and return values are modeled using dedicated program variables. `ReadToEndAsync` is modeled using a set of reads of the index and the content of the input stream, and a write to the index. As in the previous case, the await for the completion of these accesses is modeled using the `await` $*$.

```
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, which are expressed 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. Therefore, we assume that:

(1) every method invocation 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 an invocation r := `call` ... to the exit has to pass through an `await` r statement.

```
1  async method Main {
2    while *
3      r = call m;
4
5    await r;
6  }
```

```
1  async method Main {
2    r = call m;
3
4    if *
5      await r;
6  }
```

```
1  async method Main {
2    r = call m;
3    while *
4      r' = call m;
5      await r';
6    await r;
7  }
```

Fig. 7. Examples of programs to discuss syntactic constraints (m is a method whose code is irrelevant).

The first condition is used to simplify the technical exposition, while the last two conditions ensure that r stores a valid task identifier whenever an `await` r statement is executed, and that every invocation of an asynchronous method is awaited before the caller finishes. While programming languages like C# or Javascript do not enforce the latter constraint, this 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.

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.

3.2 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

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

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 ($\mathsf{w\text{-}for}(i) = j$). Similarly, a transition $(i, \mathsf{await}(*))$ corresponds to task i waiting asynchronously for the completion of an unspecified task. Non-deterministically, task i continues to the next statement, or task i is interrupted and transferred to the set of pending tasks ($\mathsf{w\text{-}for}(i)$ is set to $*$).

A transition (i, cont) represents the scheduling of the continuation of task i. There are two cases depending on whether i waited for the completion of another task j modeled explicitly in the language (i.e., $\mathsf{w\text{-}for}(i) = j$), or an unspecified task (i.e., $\mathsf{w\text{-}for}(i) = *$). In the first case, the transition is enabled only when the call stack is empty and 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 part of the semantics 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 $\rho = \mathsf{ps}_0 \xrightarrow{a_1} \mathsf{ps}_1 \xrightarrow{a_2} \ldots$ 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 $\mathbb{E}\mathsf{x}(P)$.

3.3 Traces

A trace of an execution ρ consists of several strict partial orders between the actions in ρ, in particular, a *synchronous happens-before order*, denoted by SO, which orders the actions in an execution as if all the method invocations were synchronous (even if the execution may contain asynchronous invocations), and an (asynchronous) *happens-before order*, denoted by HB, which orders actions according to typical data and control-flow constraints. A trace also contains two auxiliary partial orders: MO records the order between actions in the same method invocation, and CO is an extension of MO that additionally orders actions before a method invocation with respect to those inside that invocation. 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.

$$\frac{m \in \mathbb{M} \quad \mathsf{r} := \mathsf{x} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \ell' = \ell[r \mapsto \mathsf{g}(x), \mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pending}, \mathsf{completed}, \mathsf{c\text{-}by}, \mathsf{w\text{-}for}) \xrightarrow{(i, \mathsf{rd}(x))} (\mathsf{g}, (i, m, \ell') \circ \mathsf{stack}, \mathsf{pending}, \mathsf{completed}, \mathsf{c\text{-}by}, \mathsf{w\text{-}for})}$$

$$\frac{m \in \mathbb{M} \quad \mathsf{x} := \mathsf{le} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \ell' = \ell[\mathsf{pc} \mapsto \mathsf{next}(\ell(\mathsf{pc}))] \quad \mathsf{g}' = \mathsf{g}[x \mapsto \ell(\mathsf{le})]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pending}, \mathsf{completed}, \mathsf{c\text{-}by}, \mathsf{w\text{-}for}) \xrightarrow{(i, \mathsf{wr}(x))} (\mathsf{g}', (i, m, \ell') \circ \mathsf{stack}, \mathsf{pending}, \mathsf{completed}, \mathsf{c\text{-}by}, \mathsf{w\text{-}for})}$$

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

$$\frac{m \in \mathbb{M} \wedge \mathsf{return} \in \mathsf{inst}(\ell(\mathsf{pc})) \quad \mathsf{completed}' = \mathsf{completed}[i \mapsto \top]}{(\mathsf{g}, (i, m, \ell) \circ \mathsf{stack}, \mathsf{pending}, \mathsf{completed}, \mathsf{c\text{-}by}, \mathsf{w\text{-}for}) \xrightarrow{(i, \mathsf{return})} (\mathsf{g}, \mathsf{stack}, \mathsf{pending}, \mathsf{completed}', \mathsf{c\text{-}by}, \mathsf{w\text{-}for})}$$

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

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

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

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

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

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

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.

Definition 3.1 (Traces). The *trace* of an execution ρ is a tuple $\tau = \mathsf{tr}(\rho) = (\rho, \mathsf{MO}, \mathsf{CO}, \mathsf{SO}, \mathsf{HB})$ where MO, CO, SO, and HB denote the *method invocation order*, *call order*, *synchronous order*, and *happens-before* defined in Table 1. The set of traces of a program P is denoted by $\mathbb{T}\mathsf{r}(P)$.

4 SYNTHESIZING ASYNCHRONOUS PROGRAMS

In this section, we define the synthesis problem we investigate in this work. Essentially, given a synchronous program P and a subset of *base* methods $L \subseteq P$, our goal is to synthesize *all* asynchronous programs P_a that are equivalent to P and that are obtained by substituting every method in L with an equivalent *asynchronous* version. The base methods are 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 start by describing the space of all programs that are obtained by replacing synchronous base methods with asynchronous versions. Then, we give a precise definition of our synthesis problem. To emphasize

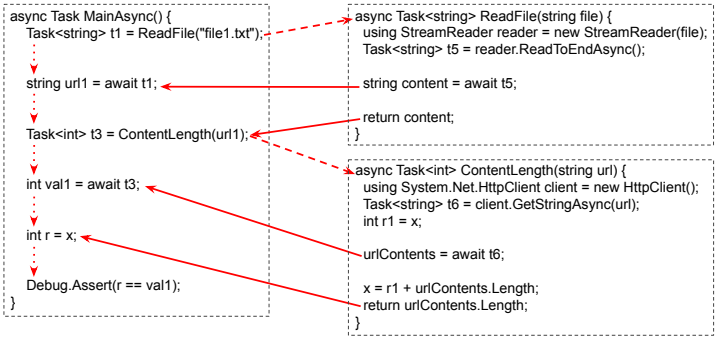


Fig. 9. A trace of an asynchronous program. Arrows between program statements denote relations between the corresponding actions in the trace.

```
1  method Main {
2    r1 = call m;
4    r2 = x;
6  }
7  method m() {
9    retVal = x;
10   x = input;
11   return;
12 }
```

```
1  async method Main {
2    r1 = call m;
3    await r1;
4    r2 = x;
6  }
7  async method m {
8    await *
9    retVal = x;
10   x = input;
11   return;
12 }
```

```
1  async method Main {
2    r1 = call m;
4    r2 = x;
5    await r1;
6  }
7  async method m {
8    await *
9    retVal = x;
10   x = input;
11   return;
12 }
```

Fig. 10. A synchronous program and all its asynchronizations.

a subset of base methods L of a program P, we use the notation $P[L]$. 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 $P[L]$ 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 (by adding the keyword `async`). This is necessary because 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 one or multiple `await` statements await r satisfying the syntactic constraints described in Section 3.1.

For instance, Fig. 10 lists a synchronous program and two possible asynchronizations, where $L = L^* = \{m\}$. Note that the only syntactic difference between the two asynchronizations is the placement of the awaits.

Async$[P, L, L_a]$ is the set of all asynchronizations of the program $P[L]$ with respect to L_a. The *strong* asynchronization, denoted by strongAsync$[P, L, L_a]$, is an asynchronization where every added `await` *immediately* follows the matching method call. The two asynchronous programs on the right of Fig. 10 represent all possible asychronizations of the synchronous program on the left. The only choices of placing the statement await r is either before or after the read of `x`. The program in the middle of Fig. 10 represents the strong asynchronization of the program on the left.

Since every asynchronous call is immediately followed by an `await`, the strong asynchronization reaches exactly the same set of program variable valuations as the original program. For instance, the strong asynchronization in Fig. 10 reaches the same set of valuations as the original program since immediately after the call to m, the matching await suspends the execution of `Main` until the call to m completes, which is exactly what happens in a synchronous execution of m.

Proposition 4.1. *For a synchronous program $P[L]$ and a set of asynchronous methods L_a as above,* $\mathbb{C}[P[L]] = \mathbb{C}[\text{strongAsync}[P, L, L_a]]$.

4.2 Problem Definition

A binary relation $\equiv$ between synchronous programs $P[L]$ and asychronizations $P_a[L_a]$ is called a *sound equivalence relation* iff it relates programs that reach the same set of program variable valuations, i.e., $P[L] \equiv P_a[L_a]$ implies $\mathbb{C}[P[L]] = \mathbb{C}[P_a[L_a]]$, for every $P_a[L_a] \in \text{Async}[P, L, L_a]$. An asynchronization $P_a[L_a]$ is called *sound w.r.t.* $\equiv$ when $P[L] \equiv P_a[L_a]$. Note that strongAsync$[P, L, L_a]$ is sound with respect to any such equivalence relation.

The problem we study in this paper is to enumerate *all* asynchronizations of a given program w.r.t. a given asynchronous library and that are sound w.r.t. a given equivalence relation.

Definition 4.2. Given a synchronous program $P[L]$, an asynchronous library L_a, and a sound equivalence relation $\equiv$, the *asychronization synthesis problem* asks to enumerate all asynchronizations in Async$[P, L, L_a]$ that are sound w.r.t. $\equiv$.

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). The enumeration algorithm we propose prioritizes asynchronizations that admit the maximal amount of parallelism from a theoretical point of view (maximizing the distance between calls and matching awaits), but it is hard to argue about their performance in a practical context.

A naive approach for solving this problem is to enumerate all asynchronizations in Async$[P, L, L_a]$ and check equivalence to the original program w.r.t. $\equiv$ for each one of them. The space of asynchronizations is exponential in the size of the program because it corresponds to all possible choices of inserting a number of `awaits` (corresponding to invocations of methods in L^*) in the code of the program's methods. Therefore, this approach leads to an enumeration algorithm that has an exponential delay complexity, i.e., the delay between two consecutive outputs is exponential in the worst case. Computing a new output would require an exponential number of calls to an oracle for checking equivalence. In the following sections, we will introduce an equivalence relation that corresponds to absence of data races and an enumeration algorithm whose delay complexity is polynomial time modulo an oracle for solving reachability (assertion checking) in sequential programs. We also present a sound but incomplete algorithm with polynomial-time delay complexity.

5 ENUMERATING SOUND ASYNCHRONIZATIONS

We describe an algorithm for solving the asynchronization synthesis problem for an equivalence relation that corresponds to absence of data races, which implies equivalence w.r.t. reachable program variable valuations. 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 of asynchronizations.

5.1 Data Race Equivalence

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 5.1 (Data Race). An asynchronous program P_a *admits data races* if there exists a trace $\tau \in \mathbb{T}\mathsf{r}(P)$ and two concurrent actions a_1 and a_2 in τ such that 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 the execution of `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`. Note that traces of *synchronous* programs can *not* contain concurrent actions, and therefore they do not admit data races.

In the following, for a data race between actions a_1 and a_2 in a trace τ, we assume that a_1 is before a_2 in the synchronous order, i.e., $(a_1, a_2) \in \mathsf{SO}$.

We introduce an equivalence relation where an asynchronization of a synchronous program P is equivalent to P if it has no data races.

Definition 5.2. An asynchronization P_a of a program P is called *data-race equivalent* to P if P_a does not admit data races.

From now on, equivalent programs will be interpreted as data-race equivalent programs. For simplicity, we will re-use $\equiv$ to denote data-race equivalence.

Absence of data races implies that the asynchronization reaches the same set of program variable valuations as the original program, and thus it is a sound equivalence relation.

Theorem 5.3. *Data-race equivalence is a sound equivalence relation.*

Proof. We have to show that for an 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 $\mathsf{ps} \in \mathbb{C}[P]$. □

5.2 Asynchronization Poset

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 define a partial order between asynchronizations which compares 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` statements, every call statement in an asynchronization $P_a[L_a] \in \mathsf{Async}[P, L, L_a]$ is associated with a *fixed* call statement in $P[L]$.

Definition 5.4. 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 the relation $\leq$ since $\mathsf{Cover}(\texttt{await } r1) = \{\}$ in the first program and $\mathsf{Cover}(\texttt{await } r1) = \{\texttt{r2 = x}\}$ in the second program.

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 any asynchronization P_a, there exists a *unique* maximal asynchronization that is smaller than P_a (w.r.t. $\leq$) and that is data-race equivalent. 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 the following hold:

- $P'_a \leq P_a$ and $P'_a \equiv P$,
- P'_a is maximal among other asynchronizations smaller than P_a and data-race equivalent to P, i.e., $\forall\, P''_a \in \mathsf{Async}[P, L, L_a].\ P''_a \equiv P$ and $P''_a \leq P_1 \Rightarrow P''_a \leq P'_a$.

Lemma 5.5. *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 a bottom element of $\leq$, there always exists a sound asynchronization smaller than P_a. Assume by contradiction that there exist two incomparable programs P^1_a and P^2_a that are both optimal asynchronizations of P relative to P_a. Let ρ^1 (resp., ρ^2) be an execution of P^1_a (resp., P^2_a) 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_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_2, such that $\mathsf{Cover}(s_1) \subset \mathsf{Cover}(s_2)$, and s_1 and s_2 match the same call in P, and (2) for every other `await` statement s'_1 in P^1_a that generates an `await` action which occurs before the `await` action of s_1 in ρ^1, there exists an `await` statement s'_2 in P^2_a matching the same call in P, such that $\mathsf{Cover}(s'_1) = \mathsf{Cover}(s'_2)$.

Let P^3_a be the program obtained from P^1_a by moving the await s_1 down (further away from the matching call) such that $\mathsf{Cover}(s_1) = \mathsf{Cover}(s_2)$. Moving an await down can only create data races between actions that occur after the execution of the matching call. Then, P^3_a contains a data race iff there exists an execution ρ of P^3_a and two concurrent actions a_1 and a_2 that occur between the action $(i, \mathsf{await}(j))$ generated by s_1 and the action $(i, \mathsf{call}(j))$ of the call matching s_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. Since the only difference between P^3_a and P^2_a 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^2_a that contains the actions a_1 and a_2 (the execution of the same statements). Also, note that a_w occurs before the action of s_1 in ρ. This implies that the action generated by the same statement s_w as a_w will occur before s_1 in ρ^1. Therefore, by the definition of s_1, s_w covers the same set of statements as an await in P^2_a. 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^2_a that contains the actions a_1 and a_2. Therefore, there exists an execution ρ' of P^2_a such that the actions a_1 and a_2 are concurrent. This implies that if P^3_a admits a data race, then P^2_a admits a data race between actions generated by the same statements. As P^2_a is data race free, we get that P^3_a is data race free as well. Since $P^1_a < P^3_a$, we get that P^1_a is not optimal, which contradicts the hypothesis. □

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)
2:     P'_a ← OptRelative(P_a);
3:     output P'_a;
4:     P ← ImmSucc(P'_a);
5:     for each P''_a ∈ P
6:         AsyncSynthesis(P''_a);
```

5.3 Enumeration Algorithm

Our algorithm for enumerating all sound asynchronizations is described in Algorithm 1 as a recursive procedure AsyncSynthesis whose initial input is the weakest asynchronization. In general, given an asynchronization $P_a \in \mathsf{Async}[P, L, L_a]$ as input, AsyncSynthesis outputs the set of all sound asynchronizations which are smaller than P_a (w.r.t. $\leq$). First, it uses the sub-procedure OptRelative to compute the optimal asynchronization P'_a of P relative to P_a, and then, calls itself recursively for all immediate successors of P'_a w.r.t. $\leq$. Formally,

$$\mathsf{ImmSucc}(P'_a) = \{P''_a \in \mathsf{Async}[P, L, L_a] \mid P'_a < P''_a \text{ and } \forall\, P'''_a \in \mathsf{Async}[P, L, L_a].\ \neg(P'_a < P'''_a \wedge P'''_a < P''_a)\}$$

Theorem 5.6. AsyncSynthesis(weakAsync$[P, L, L_a]$) *outputs the set of all asynchronizations of* $P[L]$ *w.r.t.* L_a *that are sound w.r.t.* $\equiv$.

Proof. If P_a is the weakest asynchronization of P, then the set of all sound asynchronizations of P is the set $\mathcal{A} = \{P''_a : P''_a \equiv P \text{ and } P''_a \leq P'_a\}$, where P'_a is the optimal asynchronization of P relative to P_a. Since $\leq$ is a partial order, we have that $\mathcal{A} = \{P'_a\} \cup \bigcup_{P^1_a \in \mathsf{ImmSucc}(P'_a)} \{P''_a : P''_a \equiv P \text{ and } P''_a \leq P^1_a\}$ which concludes the proof. □

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 (an implementation of OptRelative).

Theorem 5.7. *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 modifying the placement of certain `await` statements (moving them closer to the matching call statements). 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 Eliminating Data Races

Eliminating a data race (a_1, a_2) in an asynchronization P_a reduces to modifying the position of a certain `await` statement. In general, we can either move an `await` down (further away from the matching call statement), for instance in the method that executes a_1, or move an `await` up (closer to the matching call statement), for instance in the method that executes a_2. For example, let us consider the data race between `x = input` and `r2 = x` in the program on the right of Fig. 10. This data race can be eliminated by either moving the `await` $*$ in m after the write `x = input`, or

the `await r1` in `Main` before the read `r2 = x`. In the following, we will consider repairs that only consists in moving `await` statements 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 the `await` r statement), and so on. In the following, we give a precise characterization of the transformation that is sufficient for eliminating a given data race.

Note that any two racing actions have a common ancestor in the call order relation CO which is a call action. This is at least the call action of the `main` method. 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 $a_c = (i, \mathsf{call}(j))$ such that $(a_c, a_1) \in \mathsf{CO}$, $(a_c, a_2) \in \mathsf{CO}$, and for every 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 the execution of `r1 = call m` in the program on the right of Fig. 10 is the *least* common ancestor of the racing actions discussed above. The following lemma shows that this is the asynchronous call for which the matching `await` must be moved in order to eliminate a given data race. It also identifies the position where the `await` statement matching $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$ should be moved in order to eliminate 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 is predecessor of a_2 in the call order. For the program 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 6.1. *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 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), and*
- *$(a, a_w) \in \mathsf{MO}$.*

Proof. Let ρ be the execution of the trace τ. By definition, ρ ends with a configuration where the call stack and the set of pending tasks are empty. Therefore, ρ contains an action $a_w = (i, \mathsf{await}(j))$ matching a_c which is unique by the definition of the semantics. Since $(a_c, a_1) \in \mathsf{CO}$ and $(a_c, a_2) \in \mathsf{CO}$ then either a_c and a_2 occur in the same method, or there exists a call action a' in the same task as a_c such that $(a', a_2) \in \mathsf{CO}$. Then, we define $a = a_2$ in the first case, and a as the latest action in the same task as a_c such that $(a, a_2) \in \mathsf{CO}$ in the second case. We have that $(a, a_w) \in \mathsf{MO}$ because otherwise, $(a_w, a) \in \mathsf{MO}$ and $(a, a_2) \in \mathsf{CO}^*$ implies that $(a_1, a_2) \in \mathsf{HB}$ (because $(a_1, a_w) \in \mathsf{HB}$, and MO and CO are included in HB), and this contradicts a_1 and a_2 being concurrent. □

Lemma 6.1 identifies a transformation that is sufficient for eliminating a data race (a_1, a_2): moving the await statement s_w generating[2] the action a_w just before the statement s that generates the action a. This is sufficient because it ensures that every statement which is in call order after $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$[3] will be executed before a and therefore, 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. For instance, in the program on the left of Fig. 11, moving the statement `await` $r1$ after the read $r3 = y$ doesn't remove the data race in this program.

[2]Each action labels a transition in the operational semantics of a program defined in Section 3.2, and each transition corresponds to executing a particular statement. This statement is said to generate the action.
[3]We abuse the terminology and make no distinction between statements and actions.

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

```
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. 11. Examples of asynchronous programs.

The pair (s_c, s), where s_c is the call statement generating the action $\mathsf{LCA}_{\mathsf{CO}}(a_1, a_2)$, is called the *root cause* of the data race (a_1, a_2). Let $\mathsf{RepDRace}(P_a, s_c, s)$ denote the maximal asynchronization P'_a smaller than P_a w.r.t. $\leq$, such that no await statement matching s_c occurs after s on a CFG path. When the control-flow graph of the method contains branches, the construction of $\mathsf{RepDRace}(P_a, s_c, s)$ 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 Section 3.1. 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 in the middle of Fig. 11, the statement `await r1` must be moved before the read `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. 11.

6.2 Data Race Ordering

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

Given two read or write statements s and s', respectively, we use $s \prec s'$ to denote the fact that there exists an execution of P in which s is executed before the *first* time s' is executed. Then, given two actions a and a' in an execution/trace of an asynchronization of a program P, generated by two read or write statements s and s', respectively, 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[4] control-flow graph of P without taking any back edge[5].

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

Fig. 12. Non-deterministic program.

For a deterministic synchronous program (that admits a single execution), we have that $a \prec_{\mathsf{SO}} a'$ iff $s \prec s'$. However, 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

[4]We recall that the interprocedural graph is defined by taking the union of the control-flow graphs of each method and adding edges from call sites to entry nodes, and from exit nodes to return sites.

[5]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.

Fig. 12 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'.

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

Definition 6.2 (Data Race Order). Given two races (a_1, a_2) and (a_3, a_4) admitted by (possibly different) asynchronizations of a synchronous program P, we have that $(a_1, a_2) \prec_{SO} (a_3, a_4)$ iff $a_2 \prec_{SO} a_4$, or $a_2 = a_4$ and $a_1 \prec_{SO} a_3$.

Example 6.3. For the program in Fig. 13, we have the following order between data races:
$(\texttt{x = input}, \texttt{r2 = x}) \prec_{SO} (\texttt{retVal = x}, \texttt{x = r2 + 1})$ (for simplicity we use statements instead of actions) since the read `r2 = x` is executed before the write `x = r2 + 1` in the original synchronous program. However, the data races $(\texttt{x = input}, \texttt{r2 = x})$ and $(\texttt{x = input}, \texttt{r3 = x})$ are incomparable.

```
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. 13. Data race ordering.

The following lemma 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 in an iterative process.

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 $P'_a = \text{RepDRace}(P_a, s_c, s)$ 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). □

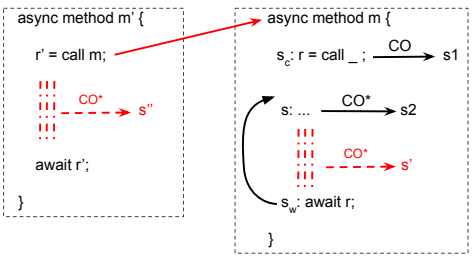


Fig. 14. An excerpt of an asynchronous program.

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 minimal data races (w.r.t. $\prec_{SO}$) iteratively until the program becomes data race free. The following theorem states that correctness of this procedure.

Theorem 6.5. *Given an asynchronization $P_a \in \text{Async}[P, L, L_a]$, 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 $P'_a = \text{OptRelative}(P_a)$ 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

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
```

moving exactly one await statement s_w in a method m of P'_a further away from the matching call s_c. Since $P^1_a \leq P_a$, 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^1_a. These actions are reachable in an execution of P^1_a 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^1_a (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 $\prec_{SO}$. Since s_w in P^1_a is placed after s, we get that a_1 and a_2 are also concurrent in that execution of P^1_a, 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.

Lemma 6.6. *For any asynchronization $P_a \in \mathsf{Async}[P, L, L_a]$, the while loop in OptRelative(P_a) does at most $|P_a|$ iterations ($|P_a|$ is the number of statements in P_a).*

The fact that data races are enumerated in the order defined by $\prec_{SO}$ 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. $\prec_{SO}$. Therefore, for a call statement s_c, let $|s_c|$ be the sum of $|\mathsf{Cover}(s_w)|$ for every await s_w matching s_c in P_a.

Lemma 6.7. *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 6.1, there exist two actions a and a' such that

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

```
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);
11

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()
30
```

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 $\perp$ 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.

where $a_w = (i, \text{await}(j))$ and $a_c = (i, \text{call}(j))$ 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 $(a, a') \in \text{MO}^*$. 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. □

7 COMPUTING ROOT CAUSES OF MINIMAL DATA RACES

We give a reduction from the problem of computing root causes of minimal data races to the reachability (assertion checking) problem in sequential (synchronous) programs. The main part of this reduction is a program instrumentation for checking if there exists a minimal data race that involves two statements given as input, 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 $\perp$) 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 7.1. *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 7.1, 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 $\perp$

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 7.2. *RootCauseMinDRace(P_a) returns the root cause of a minimal data race of P_a w.r.t. $<_{SO}$, or $\perp$ if P'_a is data race free.*

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

Theorem 7.3. *The complexity of RootCauseMinDRace is polynomial time modulo an oracle for the reachability problem in sequential programs.*

8 ASYMPTOTIC COMPLEXITY OF ASYNCHRONIZATION SYNTHESIS

In this section, we investigate the complexity of the asynchronization synthesis problem. The results in Theorem 5.7, Lemma 6.6, and Theorem 7.3 imply that outputing a new sound asynchronization reduces to a polynomial number of reachability queries in sequential programs. Therefore,

Theorem 8.1. *The delay complexity of the asynchronization synthesis problem is polynomial time modulo an oracle for the reachability problem in sequential programs.*

Since the reachability problem is PSPACE-complete for finite-state sequential programs [Godefroid and Yannakakis 2013], the results above also imply that:

Theorem 8.2. *The output[6] complexity and the delay complexity of the asynchronization synthesis problem are PSPACE for finite-state programs.*

This result is optimal in the sense that checking whether there exists a sound asynchronization which is different from the trivial strong synchronization is PSPACE-complete. The upper bound follows from Theorem 8.2 while the lower bound follows from a reduction from the reachability problem in finite-state sequential programs.

Theorem 8.3. *Checking whether there exists a sound asynchronization different from the strong asynchronization is PSPACE-complete.*

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:

- 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
- 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. □

9 COMPUTING SOUND ASYNCHRONIZATIONS USING DATA-FLOW ANALYSIS

We present a procedure for computing sound asynchronizations which is based on a bottom-up inter-procedural data-flow analysis. This procedure 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 $*$.

Given a program P, we define an abstraction $P^\#$ where every conditional `if` $<le>$ $\{S_1\}$ `else` $\{S_2\}$ is rewritten to `if` $*$ $\{S_1\}$ `else` $\{S_2\}$, and every `while` $<le>$ $\{S\}$ 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.

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

The following theorem shows that $P^{\#}$ is a sound abstraction of P in terms of sound asynchronizations it admits. Unrolling loops exactly once is sound because we assume that every asynchronous invocation in an iteration of the loop should be waited for in the same iteration (see the syntactic constraints in Section 3.1).

Theorem 9.1. *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.*

In the following, 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 and write accesses computed using a straightforward data-flow analysis, and repairs data races using the scheme presented in Section 6.1. Applying this procedure to a real programming language would require the use of an alias analysis to detect statements that may access the same memory location (this is trivial in our programming language whose purpose is to simplify the technical 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
  - 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` statement 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)

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 $\mathcal{W}_2 \cap (\mathcal{R}_1 \cup \mathcal{W}_1) \neq \emptyset$ (this denotes a conflict between sets of read/write accesses). We define a procedure OptRelative$^{\#}$ that given an asynchronization $P^{\#}_a$ proceeds as follows:

- For all methods $m \in P^{\#}_a$ in bottom-up order
  - For each statement s in the body of m from begin to end
    - If s occurs between $r :=$ `call` m' and `await` r (for some m'), and RW-var$(s) \bowtie$ CRW-var(m')
      - $P^{\#}_a \leftarrow$ RepDRace$(P^{\#}_a, r :=$ `call` $m', s)$
- Return $P^{\#}_a$

The following theorem states the correctness of OptRelative$^{\#}$. This procedure repairs data races in an order which is $\prec_{SO}$ 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

[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.

Table 2. Empirical results.

program	lines of code	methods	invocations	# repaired DRs	await	# async
Test-Azure-Boards	50	3	3	0	0	1
Azure-Remote-Monitoring	520	10	14	0	0	1
Azure-Webjobs	190	6	14	0	1	3
FritzDectCore	147	7	11	0	1	2
MultiPlatform	60	2	6	0	2	4
NetRpc	893	13	16	3	0	3
ReadFiles	101	4	9	2	0	8
ReadFile-Stackoverflow	53	2	3	1	0	1
WordpressRESTClient	133	3	10	1	0	4
UI-Stackoverflow	50	3	4	1	0	1
VBForums-Viewer	281	7	10	1	1	6

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 $\prec_{SO}$. 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 $\prec_{SO}$ than the former.

Theorem 9.2. *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 5.7 implies the following result.

Theorem 9.3. *The delay complexity of the asynchronization synthesis problem restricted to abstracted programs $P^{\#}$ is polynomial time.*

10 EXPERIMENTAL EVALUATION

In this section we present an empirical evaluation of the asynchronization enumeration approach based on data-flow analysis, described in Section 9. As benchmark, we consider a set of asynchronous C# programs extracted mostly from open-source GitHub projects. We evaluated 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 implemented the approach in Section 9 in a prototype tool that relies on the Roslyn .NET compiler platform [Roslyn 2020] to construct CFGs for methods in a given C# program (and rewrite them in SSA form). This prototype supports only a subset of the C# language and it assumes that any alias information is provided apriori. 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 any asynchronous call to a synchronous call, and then it enumerates sound asynchronizations of the obtained program. It computes the weakest asynchronization (placing all awaits just before the usage of the return value of the matching call), and then it computes optimal asynchronizations iteratively as in Algorithm 1.

Benchmark. Our evaluation uses a benchmark outlined in Table 2 that contains 8 programs extracted from open-source C# projects on GitHub (their name corresponds to a prefix of the repository name), 2 programs inspired by questions on `stackoverflow.com` about data races caused by async/await in C# (their name ends in Stackoverflow), and a manually constructed program (ReadFiles). The first three columns in Table 2 list some characteristics of our benchmark, i.e., the number of lines of code, the number of methods, and the number of method invocations.

The programs in our benchmark contain calls to 13 external methods from 5 libraries (*System.IO*, *System.Net*, *Windows.Storage*, *Microsoft.WindowsAzure.Storage*, and *Microsoft.Azure.Devices*) which have synchronous and asynchronous versions. These methods are interpreted as base methods to be substituted with asynchronous versions, and they are modeled as described in Section 3.1.

Evaluation. The last three columns in Table 2 list some data concerning the application of our tool to this benchmark. We omit running times since the tool completes under one second for each program. The column "# repaired DRs" gives the number of data races that were discovered and repaired in order to obtain the optimal asynchronization relative to the weakest asynchronization. The column "await" gives the number of statements that the awaits in the optimal asynchronization are covering *more than* in the original asynchronous program given as input. This number is zero if the await placement in the optimal asynchronization is exactly the same as in the original asynchronous program. Note that an await being moved after a call to a method m is counted as one, independently of the size of m's body. The column "# async" lists the number of sound asynchronizations outputted by our tool. In all cases, this includes the optimal asynchronization relative to the weakest asynchronization and the original asynchronous program.

These results show that our technique was able to soundly increase the distance between calls and matching awaits, which in principle, increases the amount of parallelism, or mirror the await placement in the original programs. This shows that it has the potential of becoming the basis of a refactoring tool allowing programmers to improve their usage of the async/await primitives.

11 RELATED WORK

There is large body of work 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.

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 find and 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, ensuring through syntactic pattern matching of C# namespaces that the accesses in these continuations cannot cause data races. Our work investigates a different synthesis problem with a different specification that enables a complete refactoring of a sequential program into an equivalent asynchronous program. Differently from these works, we give precise solutions for this problem and study its complexity.

Data Race Detection. There are many works that study dynamic data race detection using happens-before and lock-set analysis, or timing-based detection of data races, 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 tasks executing in parallel. [Li et al. 2019] presents a testing technique for finding data races in C# and F# programs. Their method is 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. Similarly to [Santhiar and Kanade 2017], they precompute summaries for all methods in the program in order to extract potential racy accesses. These approaches are similar to the analysis we present in Section 9, 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 for several classes of asynchronous programs. [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 (using the standard call semantics). They propose an algorithm for this verification problem which ultimately reduces to reachability in sequential programs (absence of data races) as in our work. They study a different set of asynchronous programming primitives and they do not consider the synthesis problem (or the problem of repairing programs that do not satisfy their correctness condition).

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

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. 2020. 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