Datalog-Based Scalable Semantic Diffing of Concurrent Programs

Chungha Sung
University of Southern California
Los Angeles, CA, USA

Constantin Enea
University Paris Diderot
Paris, France

Shuvendu K. Lahiri
Microsoft Research
Redmond, WA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT
When an evolving program is modified to address issues related to thread synchronization, there is a need to confirm the change is correct, i.e., it does not introduce unexpected behavior. However, manually comparing two programs to identify the semantic difference is labor intensive and error prone, whereas techniques based on model checking are computationally expensive.

To fill the gap, we develop a fast and approximate static analysis for computing synchronization differences of two programs. The method is fast because, instead of relying on heavy-weight model checking techniques, it leverages a polynomial-time Datalog-based program analysis framework to compute differentiating data-flow edges, i.e., edges allowed by one program but not the other. Although approximation is used in our method, it is sufficiently accurate due to careful design of the Datalog inference rules and iterative increase of the required data-flow edges for representing a difference. We have implemented our method and evaluated it on a large number of multithreaded C programs to confirm its ability to produce, often within seconds, the same differences obtained by human; in contrast, prior techniques based on model checking take minutes or even hours and thus can be 10x to 1000x slower.

CCS CONCEPTS
• Software and its engineering → Software verification and validation;

KEYWORDS
Concurrency, semantic diffing, change impact, static analysis, race condition, atomicity, Datalog

1 INTRODUCTION
When an evolving concurrent program is modified, often times, the sequential program logic is not changed; instead, the modification focuses on thread synchronization, e.g., to optimize performance or remove bugs such as data-races and atomicity violations. Since concurrency is hard, it is important to ensure the modification is correct and does not introduce unexpected behavior. However, manually comparing two programs to identify the semantic difference is difficult, and the situation is exacerbated in the presence of thread interactions: changing a single instruction in a thread may have a ripple effect on many instructions in other threads. Although techniques have been proposed to compute the synchronization difference, e.g., by leveraging model checkers \cite{14}, they are expensive for practice use. For example, comparing two versions of a program with 578 lines of C code takes half an hour.

To fill the gap, we develop a fast and approximate static analysis to compute such differences with the goal of reducing analysis time from hours or minutes to seconds. We assume the two programs are closely related versions of an evolving software where changes are made to address issues related to thread synchronization as opposed to the sequential computation logic. Therefore, same as in prior works \cite{14, 44}, we focus on synchronization differences. However, our method is orders-of-magnitude faster because instead of model checking we leverage a polynomial-time declarative program analysis framework which uses a set of Datalog rules to model and reason about thread interactions.

The reason why prior techniques are expensive is because they insist on being precise. Specifically, they either enumerate inter-leavings or use a model checker to ensure a semantic difference, represented as a set of data-flow edges, is allowed by one of the programs but not by the other. However, this in general is equivalent to program verification, which is an undecidable problem \cite{43}; even in cases where it is reduced to a decidable problem, the cost of model checking is too high. Our insight is that in practice, it is relatively easy for developers to inspect a given difference to determine if it is feasible; what is not easy and hence requires tool support is a systematic exploration of behaviors of the two programs to identify all possible differences in the first place. Unfortunately, developing such a tool is a non-trivial task; for example, the naive approach of comparing individual thread inter-leavings would not work due to the often exponential blowup in the number of inter-leavings.

Our method avoids the problem by being approximate in that it does not enumerate inter-leavings. This also means infeasible behaviors are sometimes included. However, our approximation is carefully designed to take into consideration the program semantics
most relevant to thread interaction. Furthermore, the approximation can be refined by iteratively increasing the number of data-flow edges used to characterize a synchronization difference. We shall show through experiments that our fast and approximate analysis method does not lead to overly inaccurate results. To the contrary, the synchronization differences reported by our method closely match the ones identified by human. Compared to the prior technique based on model checking, which often takes minutes or even hours, our method can be 10x to 1000x faster.

Figure 1 shows the overall flow of our method. The input consists of two versions of a concurrent program: \( P_1 \) is the original version, \( P_2 \) is the changed version, and patch info represents their syntactic difference, e.g., information about which instructions are added, removed or modified. The output consists of a set of differences, each of which is represented by a set of data-flow edges allowed in one of the programs but not the other. When data-flow edges are allowed in \( P_1 \) but not \( P_2 \), for example, they represent a removed behavior. Conversely, when data-flow edges are allowed in \( P_2 \) but not \( P_1 \), they represent a new behavior introduced by the change.

Our method first generates a set of \( \mu \mathcal{Z} \) Datalog facts that encode the structural information of the control flow graphs. These facts are then combined with inference rules that codify the analysis algorithm. When the combined program is fed to a \( \mu \mathcal{Z} \) Datalog solver, the resulting fixed point contains new relations (facts) that represent the analysis result. Specifically, it contains data-flow edges that may occur in each program. By comparing data-flow edges from the two programs, we can identify the semantic differences.

Since program verification is undecidable in general, and with concurrency, it is undecidable even for Boolean programs [43], approximation is inevitable. Our method makes two types of approximations. The first one is in checking the feasibility of data-flow edges. The second one is related to the number of data-flow edges used to characterize a difference, also referred to as the rank of an analysis [14]. Although in the worst case, a precise analysis means the rank needs to be as large as the length of the execution, we restrict it to a small number in our method because prior research [12, 40] shows that concurrency bugs often can be exposed by executions with a bounded number of context switches.

Since our method is approximate in nature, the usefulness depends on how close it approaches the ground truth. Ideally, we want to have few false positives and few false negatives. Toward this end, we choose to stay away from the tradition of insisting the analysis being either sound or complete when one cannot have both. For a concurrent program, being sound often means existential abstraction: a data-flow edge is considered feasible (in all interleavings) if it is feasible in an interleaving, and being complete often means universal abstraction: a data-flow edge is considered feasible only if it is feasible in all interleavings. Both cases result in extremely coarse-grained approximations, which in turn lead to numerous false positives or false negatives. Instead, we want to minimize the difference between our analysis result and the ground truth.

We have implemented our method in a tool named EC-Diff, which uses LLVM [9] as the front-end and \( \mu \mathcal{Z} \) [24] in Z3 as the Datalog solver. We evaluated EC-Diff on 47 multithreaded programs with 13,500 lines of C code in total. These are benchmarks widely used in prior research [1–7, 11, 13, 23, 38, 47, 50–52, 55]: some illustrate real concurrency bug patterns [52] and the corresponding patches [29] while others are applications from public repositories. We applied EC-Diff to these benchmarks while comparing with the prior technique of Bouajjani et al. [14]. Our results show that EC-Diff can detect, often in seconds, the same differences identified by human. Furthermore, compared to the prior technique based on model checking, EC-Diff is 10x to 1000x faster.

To summarize, this paper makes the following contributions:

- We propose a fast and approximate analysis based on a polynomial-time declarative program analysis framework to compute synchronization differences.
- We show why our approximate analysis is reasonably accurate due to the custom-designed inference rules and iterative increase of the number of data-flow edges.
- We implement our method in a practical tool and evaluate it on a large number of benchmarks to confirm its high accuracy and low overhead.

The remainder of the paper is as follows. First, we motivate our work using examples in Section 2. Then, we provide the technical background in Section 3 before presenting our analysis method in Section 4. This is followed by our procedures for interpreting the analysis result and optimizing performance in Section 5. We present our experimental results in Section 6. Finally, we review the related work in Section 7 and give our conclusions in Section 8.

2 MOTIVATION

We use examples to motivate the need for conducting a differential analysis. Programs used in these examples illustrate common bug patterns (also used during our experiments in Section 6). In each example, there are two program versions: the original one may violate a hypothetical assertion and the changed one avoids it. These assertions are hypothetical (added for illustration purposes only) in the sense that our method does not need them to operate.

2.1 The First Example

Fig. 2(a) shows a two-threaded program where the shared variable \( x \) is initialized to 0. The assertion at Line 3 may be violated, e.g., when \( \text{thread1} \) executes the statement at Line 2 right after \( \text{thread2} \) executes the statement at Line 5. The reason is because no synchronization operation is used to enforce any order.

Assume the developer identifies the problem and patches it by adding locks (Figure 2(b)), the assertion violation will be avoided.

To see why this is the case, consider the data-flow edge from Line 5 to Line 2: due to the critical sections enforced by lock-unlock pairs, the load of \( x \) at Line 2 is not affected by the store of \( x \) at Line 5. For example, if the critical section containing Line 5 is executed first, the subsequent \( \text{unlock}(a) \) must be executed before the \( \text{lock}(a) \) in \( \text{thread1} \), which in turn must be executed before Line 1 and Line 2.
Since the store of $x$ at Line 1 is the most recent, the load of $x$ at Line 2 will get its value, not written at Line 5.

Thus, the allowed data-flow edges are as follows: $RF((L4, L2)$ and $RF((L5, L1)$ for the original program, and $RF((L4, L2)$ and $RF((L5, L1)$ for the changed program. This notion of comparing concurrent executions was introduced by Shasha and Snir [44] and extended by Bouajjani et al. [14], although in both cases, enumeration or model checking techniques were used. In our work, the goal is to avoid such heavyweight analyses while maintaining sufficient accuracy.

In addition to $RF$ edges, there are other types of relations considered during our analysis, including program order, inter-thread order imposed by thread create, join, signal-wait as well as store-store order. Nevertheless, when interpreting the final results, we focus on differences in the $RF$ edges because they affect the externally observable behavior of a program, e.g., characterized by assertions and other reachability properties.

### 2.2 The Second Example

Fig. 3 shows a more sophisticated example: the use of signal-wait, which is often difficult for static analyzers. Since the variable $x$ is initialized to 0, when the critical section in thread1 is executed before thread2, the load of $x$ at Line 1 will get the value 0, which leads to the assertion violation in Fig. 3(a). Assume the intended behavior is for thread2 to complete first, an inter-thread execution order must be enforced, e.g., by using the signal-wait pair shown in Fig. 3(b). The assertion violation is avoided because the load of $x$ at Line 1 can only read from the store of $x$ at Line 5.

To correctly deploy the signal-wait pair, a variable named cBool needs to be added. If the operating system voluntarily schedules thread2 first, thread1 needs to be aware – by checking the value of cBool – and then skips the execution of wait; otherwise, wait may get stuck because the corresponding signal has already been fired (and lost). But if thread1 is executed first, since cBool has not been set, it will invoke wait which forces the corresponding signal to be sent.

As for the data-flow edges, we can see that $RF((L5, L1)$ and $RF((L3, L4)$ are allowed in the original program, but only $RF((L5, L1)$ is allowed in the changed program. $RF((L3, L4)$ is not allowed because Line 4 must happen before Line 5, Line 5 must happen before

**Figure 2:** Example programs with synchronization differences (lock-unlock).

**Figure 3:** Example programs with synchronization differences (signal-wait).

**Figure 4:** Analysis steps for programs in Figs. 3(a) and 3(b).
For now, consider the steps of computing synchronization differences for the programs in Fig. 2 and Fig. 3, which are outlined by the tables in Fig. 4.

First, our method computes must-happen-before (mustHB) edges, which represent the execution order of two instructions respected by all thread interleavings. From mustHB, our method computes may-happen-before (mayHB) edges, which represent the execution order respected by some interleavings, e.g., thread context switches not contradicting to mustHB. From mayHB, our method computes MayRF edges, which represent data flows (over shared variables) from store instructions to the corresponding load instructions.

The MayRF edges are over-approximated in that, if an edge is included in MayRF, the corresponding data flow may occur in an execution. But if an edge is not included in MayRF, we know for sure the corresponding data flow is definitely infeasible. For example, in Fig. 4, MayRF has four edges for Fig. 2(a) but only three edges for Fig. 2(b). RF (L5, L2) is no longer allowed in the changed program, indicating it is a difference between the two programs.

For the example in Fig. 3, we compute mustHB based on the sequential program order and, in Fig. 3(b), the inter-thread execution order imposed by signal-wait. Then, from mustHB we compute mayHB, which includes edges in mustHB and more. For Fig. 3(a), since there is no restriction on the inter-thread execution order, all pairs of events are included, whereas for Fig. 3(b), there is only one-way data flow. Finally, we compute MayRF based on mayHB. There are three edges for Fig. 3(a) but only two for Fig. 3(b).

2.4 The Rank of an Analysis

When comparing MayRF in these two examples, we identify the difference as edges allowed in only one of the two programs, such as RF (L5, L2) in Fig. 2 and RF (L3, L4) in Fig. 3.

However, even if MayRF edges are allowed individually, they may not occur in the same execution. For example, RF (L5, L1) and RF (L3, L4) in Fig. 3(a) cannot occur together because, otherwise, they form a cycle together with the program order edges. Our method has inferences rules designed to check if two or more dataflow edges can occur together—this is referred to as the rank [14].

With the notion of rank, we can capture ordered sets of MayRF edges, as opposed to individual MayRF edges. Thus, even if the MayRF relation remains the same, there may be differences of high ranks: two or more edges from MayRF may occur together in P1 but not in P2. We will present our method for checking such differences in Section 5 following the baseline procedure in Section 4.

3 PRELIMINARIES

3.1 Partial Trace Comparison

To compare the synchronizations of two concurrent programs, we use the notion of partial trace introduced by Shasha and Snir [44] and extended by Bouajjani et al. [14]. Let P be a program and S be the set of global variables shared by threads in P. For each x ∈ S, let W(x) denote a store instruction and R(x) denote a load instruction. Let I be the set of all instructions in the program. Any binary relation over these instructions is a subset of I × I.

For example, so ⊆ I × I is a relation that orders the store instructions; W1(x) < W2(x) means W1 ∈ I is executed before W2 ∈ I. Thus, in Fig. 2(a), (L1, L4), (L4, L1), (L1, L5), (L5, L1), (L4, L5) belong to so, but (L5, L4) does not belong to so because it is not consistent with the program order.

Similarly, rf is a relation between load and store instructions. In Fig. 2(a), we have (L4, L2) and (L5, L2) in rf, meaning the load at Line 2 may read from values written at Lines 4 and 5. Given so and rf, we define sets as a set of subsets of rf ∪ so, where each element ss ∈ sets has at most k edges.

Edges in ss are from either rf or so—they capture the abstract trace. The number k, which is called the rank [14], is bounded by the length of the trace.

Definition 1 (Abstract Trace with Rank k). An abstract trace with rank k is a tuple T = (so, rf, sets, k), where so ⊆ {W1(x) × W2(x) | W1 ∈ I, W2 ∈ I, and W1 < W2 in some execution trace}, rf ⊆ {R(x) | W ∈ I and R ∈ I}, and sets ⊆ {ss ⊆ rf ∪ so | |ss| ≤ k}.

Given the abstract traces T1 and T2 of two programs P1 and P2, respectively, we define their difference as D = (Δ12, Δ21), where Δ12 = T1 \ T2 and Δ21 = T2 \ T1. Next, we define what it means for T1 to be a refinement of T2, denoted T1 ⊆ T2.

Definition 2 (Abstract Trace Refinement). Given two abstract traces T1 = (so1, rf1, sets1, k) and T2 = (so2, rf2, sets2, k), we say T1 is a refinement of T2, denoted T1 ⊆ T2, if and only if so1 ⊆ so2, rf1 ⊆ rf2, and sets1 ⊆ sets2.

That is, when T1 ⊆ T2, the abstract behavior of P1 is covered by that of P2. And the difference (T2 \ T1) is characterized by so1 \ so2, rf2 \ rf1, and sets2 \ sets1. Finally, if the abstract traces of P1 and P2 refine each other, we say they are rank-k equivalent.

Although comparison of abstract traces involves so and rf, when reporting the differences, we focus on the rf edges only because they directly affect the observable behaviors of the programs. In contrast, store-store ordering (so) may not be observable unless they also affect the read-from (rf) edges.

3.2 Datalog-Based Analysis

Datalog is a logic programming language but in recent years has been widely used for declarative program analysis [10, 15, 16, 21, 22, 37, 48, 56]. The main advantage is that a Datalog program is polynomial-time solvable and the corresponding fixed-point computation maps naturally to fixed-point computations in program analysis algorithms. In this context, structural information of the program is represented as relations called the facts, while the fixed-point algorithm is expressed as recursive relations called the inference rules.

Consider a relation named PO(a, b), which represents the program order of two immediate adjacent instructions a and b, while HB(c, d) means c must happen before d. First, we write down the Datalog facts based on the CFG structure:

PO(x2, x1), PO(x1, x3), PO(x2, x3), PO(x2, x4), PO(x1, x4), PO(x1, x5).

Then, we write down the Datalog inference rules:

HB(a, b) ← PO(a, b)
HB(c, e) ← HB(c, d) ∧ HB(d, e)

Here, the left arrow (←) separates the inferred Datalog facts on the left-hand side from the existing Datalog fact(s) on the right-hand side. The first rule says the program-order relation implies the must-happen-before relation. The second rule says the must-happen-before relation is transitive.
A Datalog solver, based on the above facts and rules, will compute the maximal set of edges for the HB relation. By sending a query to the Datalog solver, one may confirm that HB(s1, s2) indeed holds whereas HB(s2, s3) does not hold.

4 CONSTRAINT-BASED SYNCHRONIZATION ANALYSIS

In this section, we present our method for computing abstract traces of a single program. In the next section, we leverage the abstract traces of two programs to compute their differences.

First, we define the elementary relations that can be constructed directly from the CFG of a program.

- St(s1, th1): Statement s1 resides in Thread th1
- Po(s1, s2): Statement s1 is before s2 in a thread
- Dom(s1, s2): Statement s1 dominates s2 in a thread
- PostDom(s1, s2): s1 post-dominates s2 in a thread
- ThrdCreate(th1, s1, th2): Thread th1 creates th2 at s1
- ThrdJoin(th1, s1, th2): Thread th1 joins back th2 at s1
- CondWait(s1, th1): s1 waits for condition variable v1
- CondSignal(s1, th1): s1 sends condition variable v1
- Load(s1, v2): Statement s1 reads from variable v1
- Store(s1, v2): Statement s1 writes to variable v1
- IncS(s1, l1): s1 resides in a critical section guarded by lock(l1)
- SameCS(s1, s2, l1): s1 and s2 are in the same critical section guarded by l1
- DiffCS(s1, s2, l1): s1 and s2 are in different critical sections guarded by l1

While traversing the CFG to compute the Po, Dom, and PostDom relations, we take loops into consideration. For example, two instructions involved with the same loop may not have a Dom or PostDom relation, but an instruction outside the loop can have a Dom or PostDom relation with an instruction inside the loop.

Next, we define inference rules for computing new relations such as MayHB, MustHB, and MayRF.

4.1 Rules for Intra-thread Dependency

To capture the execution order of instructions, we define the following relations: \( \text{MaxHB}(s1, s2) \) means s1 may happen before s2 in some execution, and \( \text{MustHB}(s1, s2) \) means s1 happens before s2 in all executions when both occur. Since the program order in each thread implies the execution order, we have the following rule:

\[
\text{MustHB}(s1, s2) \leftarrow \text{Po}(s1, s2)
\]

In this work, we assume sequential consistency but Datalog is capable of handling weaker memory models [32] as well.

By definition \( \text{MustHB} \) implies \( \text{MayHB} \), which means

\[
\text{MaxHB}(s1, s2) \leftarrow \text{MustHB}(s1, s2)
\]

4.2 Rules for Inter-thread Dependency

When a parent thread th1 creates a child thread th2 at the statement s1, e.g., by invoking pthread_create, any statement s2 in the child thread must occur after s1.

\[
\text{MustHB}(s1, s2) \leftarrow \text{ThrdCreate}(th1, s1, th2) \wedge \text{St}(s1, th1)
\]

Similarly, when a parent thread th1 joins back a child thread th2 at s1, any statement s2 in th2 must occur before s1.

\[
\text{MustHB}(s2, s1) \leftarrow \text{ThrdJoin}(th1, s2, th2) \wedge \text{St}(s1, th1)
\]

4.3 Rules for Signal-Wait Dependence

When a condition variable c is used, e.g., through signal(c) and wait(c), it imposes an execution order.

\[
\text{MustHB}(s1, s2) \leftarrow \text{CondSignal}(c, s1) \wedge \text{CondWait}(c, s2)
\]

However, the rule needs to be used with caution. In practice, wait(c) is often wrapped in an if-condition as shown in Figure 5(b).

To be conservative, our method analyzes the control flow of these threads and applies the above rule only after detecting the usage pattern. Since our method does not analyze the concrete values of any shared variables, it does not check if the if-condition is valid. Also, developers may use condition variables in a different way. Thus, in our experiments (Section 6), we evaluated the impact of this conservative approach—assuming the if-condition is always valid—to confirm it does not lead to significant loss of accuracy.

4.4 Ad Hoc Synchronization

We handle ad hoc synchronization similar to signal-wait. Fig. 5 shows an example where cond is a user-added flag initialized to 0. The busy-waiting in thread2 ensures that a=1 always occurs before x=a. By traversing the CFGs of these threads, we can identify the pattern; this is practical since the number of usage patterns is limited. After that, we add a \( \text{MustHB} \) edge from \( \text{cond=\text{true}} \) to while(!cond). This is similar to adding \( \text{MustHB} \) edges for CondWait and CondSignal. As a result, we can decide the read-from edge between x=a and the initialization of a is infeasible.

4.5 Transitive Closure

Since \( \text{MustHB} \) is transitive, we use the following rule to compute the transitive closure:

\[
\text{MaxHB}(s1, s2) \leftarrow \text{MaxHB}(s1, s3) \wedge \text{MaxHB}(s3, s2)
\]

When instructions in concurrent threads are not ordered by \( \text{MustHB} \), we assume they may occur in any order:

\[
\text{MaxHB}(s1, s2) \leftarrow \text{St}(s1, th1) \wedge \text{St}(s2, th2) \wedge \neg \text{MaxHB}(s2, s1)
\]

The \( \text{MayHB} \) relation is also transitive:

\[
\text{MaxHB}(s1, s2) \leftarrow \neg \text{MaxHB}(s1, s3) \wedge \text{MaxHB}(s3, s2)
\]

4.6 Lock-Enforced Critical Section

For critical sections based on lock-unlock, we introduce rules based on access patterns. First, we compute \( \text{CoveredStore}(s1, v1, l1) \), meaning the store in s1 is covered by a subsequent store in the same critical section. Consider \( l_k(a) \rightarrow W_1(v) \rightarrow W_2(v) \rightarrow \neg l_k(a) \), where \( W_1(v) \) is a covered store and thus not visible to reads in other critical sections protected by the same lock.

\[
\text{CoveredStore}(s1, v1, l1) \leftarrow \text{St}(s1, th1) \wedge \text{St}(s2, v1) \wedge \neg \text{MaxHB}(s2, s1) \wedge \text{SameCS}(s1, s2, l1)
\]

Similarly, \( \text{CoveredLoad}(s2, v1, l1) \) means the load of \( v1 \) in \( s2 \) is covered and thus can only read from a preceding store in the same
Consider $1k(a) \rightarrow W(v) \rightarrow R(v) \rightarrow un1k(a)$ as an example: $R(v)$ is covered by $W(v)$ and thus cannot read from stores in other critical sections protected by the same lock.

4.7 Read-from Relation

Finally, we compute $NoRf(s_1, s_2)$ which means the read-from edge between $s_1$ and $s_2$ is infeasible.

$$NoRf(s_1, s_2) \leftarrow Store(s_1, v_1) \land Store(s_1, v_1) \land Load(s_2, v_1)$$

That is, in $W(x) \rightarrow W(x) \rightarrow R(x)$, the first store cannot be read by the load. In addition to this generic rule, we have two more inference rules:

$$NoRf(s_1, s_2) \leftarrow Store(s_1, v_1) \land Load(s_2, v_2) \land \text{MustHb}(s_1, s_2)$$

This rule means if one store may happen before one load, the load is covered, and the store is in a different critical section, the load cannot read from the store. This is because another store will overwrite the value to be read.

$$NoRf(s_1, s_2) \leftarrow Store(s_1, v_1) \land Load(s_2, v_2) \land \text{MaxHb}(s_1, s_2)$$

This rule means if a store is covered, i.e., overwritten by a subsequent store, the store cannot reach to any load in another critical sections protected by the same lock.

We also compute $\text{MaxRf}(s_1, s_2)$ which means the load in $s_2$ may read from the store in $s_1$.

$$\text{MaxRf}(s_1, s_2) \leftarrow Store(s_1, v_1) \land Load(s_2, v_1) \land \text{MaxHb}(s_1, s_2)$$

5 COMPUTING THE DIFFERENCES

In this section, we show how to compare abstract traces of the two programs to identify the differences.

5.1 Symmetric Difference

Fig. 6 shows the Venn diagram of our method for computing the differences when given the abstract traces of two programs. The actual behaviors of programs $P_1$ and $P_2$ are represented by the circles with solid lines. The approximate behaviors, in the form of abstract traces $\tilde{T}_1$ and $\tilde{T}_2$, are represented by the circles with dashed lines. Conceptually, the symmetric difference is computed based on $\Delta_{12} = \tilde{T}_1 \setminus \tilde{T}_2$ and $\Delta_{21} = \tilde{T}_2 \setminus \tilde{T}_1$, and for each is presented as pink-colored region in Fig. 6 (left and right). The details of them are presented in the remainder of this section.

To compute the difference, we define two relations $\text{DiffP}1$ and $\text{DiffP}2$ and rules for computing them:

$$\text{DiffP}1(s_1, s_2) \leftarrow \text{MaxRf}(s_1, s_2, P_1) \land \text{MaxRf}(s_1, s_2, P_2)$$

$$\text{DiffP}2(s_1, s_2) \leftarrow \text{MaxRf}(s_1, s_2, P_2) \land \text{MaxRf}(s_1, s_2, P_1)$$

DiffP1 represents edges that may happen in $P_1$ but not in $P_2$. Similarly, DiffP2 represents edges that may happen in $P_2$ but not in $P_1$. If DiffP1 is not empty, there are more behaviors in $P_1$; and if DiffP2 is not empty, there are more behaviors in $P_2$.

Since the Datalog solver may enumerate all possible $\text{MaxHb}$ edges (used to compute $\text{MaxRf}$), and the number of $\text{MaxHb}$ edges increases rapidly as the program size increases, we need to reduce the computational overhead. Our insight is that, since we are only concerned with synchronization differences in the end, as opposed to behaviors of the sequential computation, we can restrict our analysis to instructions that access global variables. Toward this end, we define a new relation named $\text{Access}(v_1, s_1)$ which means $s_1$ accesses a global variable $v_1$, and use it to guard the inference rules for $\text{MaxHb}$ (and hence $\text{MustHb}$). It forces the Datalog solver to consider only global accesses, which reduces the computational overhead without losing accuracy. We demonstrate the effectiveness of this optimization using experiments in Section 6.

5.2 Differences at Higher Ranks

The rules so far use individual read-from edges to characterize the differences, which is equivalent to rank-1 analysis [14], but some programs may not have rank-1 difference but have differences of higher ranks. To detect them, we need to compute ordered sets of data-flow edges allowed in one program but not in the other.

To be specific, for rank-2, we extend the $\text{MaxRf}$ relation, which was defined over two instructions (an edge), to $\text{MaxRfs}$ defined over four instructions, to represent an ordered set of (two) read-from edges. Similarly, we extend the $\text{NoRf}$ relation to $\text{NoRfs}$, which is also defined over four instructions to represent an ordered set of (two) read-from edges.

Previously, $\text{NoRf}(s_1, s_2)$ means there is no execution trace where the store $s_1$ can be read by the load $s_2$, whereas $\text{MaxRf}(s_1, s_2)$ means there may exist some execution trace that allows the read-from edge $(s_1, s_2)$. Similarly, $\text{NoRfs}(s_1, s_2)$ means there is no execution trace where the two read-from edges $(s_1, s_2)$ and $(s_3, s_4)$ occur together and in that order; and $\text{MaxRfs}(s_1, s_2)$ means there may exist some execution trace that allows the two read-from edges to occur together and in that order.

First, we present our rules for computing $\text{NoRfs}$, which in turn is used to compute $\text{MaxRfs}$. Since it is not possible to enumerate all scenarios due to theoretical limitations, we resort to the most common scenarios. Nevertheless, we guarantee that $\text{NoRfs}$ is an under-approximation, and the corresponding $\text{MaxRfs}$ is an over-approximation.

$$\text{NoRfs}(s_1, s_2, s_3, s_4) \leftarrow \text{MaxRf}(s_1, s_2) \land \text{MaxRf}(s_3, s_4)$$

This rule is obvious because, as in Fig. 7 (left), in the same execution trace a load $(s_3)$ cannot read from two different stores $(s_1, s_2)$. $\text{MaxRfs}(s_1, s_2, s_3, s_4)$ includes all possible execution traces where the two read-from edges $(s_1, s_2)$ and $(s_3, s_4)$ occur together and in that order.
This rule is also obvious because, as shown in Fig. 7 (right), if the two read-from edges form a cycle together with the must-happen-before edges, they lead to a contradiction.

\[
\text{NoRFs}(s_1, s_2) \iff \text{SameCS}(s_1, s_2, l_1) \land \text{SameCS}(s_2, s_3, l_2) \\
\land \text{DiffCS}(s_1, s_2, l_1)
\]

This rule is related to lock-unlock pairs. The rationale behind it can be explained using the diagram in Fig. 8 (left). Due to the lock-unlock pairs, there are only two possible interleavings: (1) if \(s_1\) happens before \(s_2, s_4\) must happen before \(s_3\) and \(s_5\), which contradicts to the read-from edge \((s_3, s_4)\); (2) if \(s_3\) happens before \(s_4\), \(s_2\) must happen before \(s_1\), which contradicts to the read-from edge \((s_1, s_2)\). Thus, the read-from edges cannot occur in the same execution trace.

Next, we define another rule related to lock-unlock pairs. In this rule, we use \(\text{PostDom}(s_5, s_2)\) to mean, after \(s_2\) is executed, \(s_3\) is guaranteed to be executed as well.

\[
\text{NoRFs}(s_2, s_4) \iff \text{Store}(s_2, s_2) \land \text{PostDom}(s_5, s_2) \\
\land \text{DiffCS}(s_2, s_4, l_4) \land \text{SameCS}(s_2, s_3, l_3)
\]

The rationale behind this rule can be explained using the diagram in Fig. 8 (right). Here, the loads and stores access the same variable. If the read-from edge \((s_1, s_2)\) is ahead of \((s_3, s_4)\) in the same execution trace, the store in \(s_3\) contradicts to the read-from edge \((s_1, s_2)\).

Finally, we compute MayRFs based on NoRFs:

\[
\text{MayRFs}(s_1, s_2, s_3, s_4) \iff \neg \text{NoRFs}(s_1, s_2, s_3, s_4)
\]

It means the read-from edges \((s_1, s_2)\) and \((s_3, s_4)\) may occur together and in that order in some execution traces. With MayRFs, we compute differences \((\text{DistP1} \land \text{DiffP2})\) by replacing MayRF with MayRFs. Our method for computing differences of rank 3 or higher are similar, and we omit the details for brevity.

### 5.3 Example for Rank-2 Analysis

Fig. 9 shows an example that illustrates the rank-2 analysis. Here, \(\text{thread1}\) sets \(t\) to 0 and \(y\) to 1 before creating \(\text{thread2}\). Due to lock-unlock pairs, the assertion cannot be violated in Fig. 9(a). However, if the lock-unlock in \(\text{thread1}\) is removed as in Fig. 9(b), the assertion may be violated because, in between Lines 4 and 5, there may be a context switch which was not allowed previously.

However, the synchronization difference cannot be captured by any individual MayRF edge. In fact, the table in Fig. 10 shows that the two programs have the same set of MayRF edges. In particular, since there are two stores of \(x\), the load at Line 2 may read from both Line 1 and Line 5.

To capture the difference, we need rank-2 analysis.

- Assume \(\text{RF}(L_1, L_4)\) occurs first, meaning \(\text{thread2}\) acquires the lock and thus prevents \(\text{thread1}\) from acquiring the same lock until \(\text{thread2}\) exits the critical section. It means the store at Line 5 will set \(x\) to 2. Therefore, the load of \(x\) at Line 2 will have to read from Line 5, not from Line 1. In other words, \(\text{RF}(L_1, L_2)\) cannot occur after \(\text{RF}(L_1, L_4)\) in the same execution.
- Assume \(\text{RF}(L_1, L_2)\) occurs first and \(\text{thread2}\) will not be executed until \(\text{thread1}\) finishes. In this case, \(\text{RF}(L_1, L_4)\) is allowed since no store of \(x\) is in \(\text{thread1}\).

As a result, the program in Fig. 9(a) allows the ordered set \([\text{RF}(L_1, L_2), \text{RF}(L_1, L_4)]\) but not the ordered set \([\text{RF}(L_1, L_4), \text{RF}(L_1, L_2)]\).

However, the program in Fig. 9(b) allows the ordered set \([\text{RF}(L_1, L_4), \text{RF}(L_1, L_2)]\) as well, due to the removal of the lock-unlock pairs in \(\text{thread1}\). Specifically, when \(\text{RF}(L_1, L_4)\) occurs at the start of an execution, \(\text{thread1}\) may execute Line 2 before \(\text{thread2}\) execute Line 5, which allows Line 2 to read the value of \(x\) from Line 1.

Our steps of conducting the rank-2 analysis, based on inference rules presented so far, are shown in Fig. 10. There is no difference in the MayRF sets; however, when comparing the ordered set of MayRF edges, we can still see the difference. To support this analysis, we apply the aforementioned inference rules of rank 2, which checks the existence of \((1,4) \rightarrow (1,2)\).
6 EXPERIMENTS

We have implemented the method in a tool named EC-Diff, which uses LLVM [9] as the frontend and $\mu Z$ [24] in Z3 as the Datalog solver at the backend. Specifically, we use Clang/LLVM to parse the C/C++ code of multithreaded programs and construct the LLVM intermediate representation (IR). Then, we traverse the LLVM IR to generate program-specific Datalog facts. These Datalog facts, when combined with a set of program-independent inference rules, form the entire Datalog program. Finally, the $\mu Z$ Datalog solver is used to solve the program, which repeatedly applies the rules to the fact until a fixed point is reached. By querying relations in the fixed point, we can retrieve the analysis result.

6.1 Experimental Setup

We used two sets of benchmarks in our experiments. The first set of benchmarks consists of 41 multithreaded programs, which previously [29] have been used to illustrate concurrency bug patterns found in real applications [1–7, 11, 13, 23, 38, 51, 52]. With these programs, our goal is to evaluate how well the various types of concurrency bugs are handled by our method, and how our results compare to that of the prior technique based on model checking [14]. For these benchmarks, the prior technique is not able to soundly instrument all applications. Therefore, we manually insert assertions to be checked later by the CBMC bounded model checker for detecting only one different edge.

The second set of benchmarks consists of 6 medium-sized applications from open-source repositories; they have also been used previously [49, 52] to evaluate testing and automated program repair tools. Similarly, we are not able to apply the prior technique [14] because it has limitations to instrument large size programs and it is impossible for us to manually insert assertions. Nevertheless, we can evaluate how efficient our new method EC-Diff is on these real applications. In total, our benchmarks has 13,500 lines of C code.

For each benchmark program, there are two versions, one of which is the original program and the other is the changed program. These changed programs are patches collected from various sources: some are from benchmarks used in prior research on testing [49, 52] and repair [29], whereas others are from benchmarks used in differential analysis [14]. We also created four programs, case1-4, to illustrate motivating examples used throughout this paper. These benchmark programs, together with our experimental data, the LLVM-based tool, as well as data obtained from applying the prior technique [14], have been made available online.

Our experiments were designed specifically to answer the following research questions:

- Is our new method, based on a fast and approximate static analysis as opposed to heavy-weight model checking techniques, accurate enough for identifying the actual synchronization differences in the benchmark programs?
- Is our new method significantly more efficient, measured in terms of the analysis time, than the prior technique based on model checking?

In all these experiments, we used a computer with an Intel Core i5-4440 CPU @ 3.10 GHz x 4 CPUs with 12 GB of RAM, running the Ubuntu-16.04 LTS operating system.

6.2 Results on the First Set of Benchmarks

Table 1 shows our results on the first set of benchmarks, with 41 programs illustrating common bug patterns. Columns 1 and 2 show the name and the number of lines of C code. Column 3 shows the number of threads. Column 4 shows the type of bug illustrated by the program. Specifically, Sync. means the bug is due to misuse of locks, and thus to repair it, some lock-unlock pairs have been added, removed or modified; Cond. means the bug is due to misuse of condition variables, and thus to repair it, some signal-wait pairs have been added, removed or modified; Th. Order means the bug is related to thread creation and join and thus involves ThrdJoin or ThrdCreate; and Order. means the bug is related to ordering of instructions imposed by ad-hoc synchronization. Note that, in each of these benchmarks, there is some synchronization difference. The remaining columns show the statistics reported by EC-Diff as well as the prior technique [14]. Specifically, Column 5 shows if EC-Diff detected the synchronization difference. Column 6 shows at which rank our analysis is conducted (Section 5): we iteratively increase the rank starting from 1, until a synchronization difference is detected. To be efficient, we bound the rank to 3 during our evaluation. Columns 7 and 8 show the number of differences in $\Delta_{12} = \overline{T_1} \setminus \overline{T_2}$ and $\Delta_{21} = \overline{T_2} \setminus \overline{T_1}$. For a rank-1 analysis, it is the number of read-from edges; for a rank-2 or rank-3 analysis, it is the number of ordered sets of read-from edges. The next two columns show the total number of MayHs edges (used to compute MayRs) in $P_1$ and $P_2$, respectively.

For each benchmark, we limit the run time to one hour. Our results show EC-Diff often finishes each benchmark in a second whereas the prior technique can take up to 2,384 seconds (r18169-2). In total, EC-Diff took less than 16 seconds whereas the prior technique took more than 3 hours. In terms of accuracy, except for one program, EC-Diff detected all the synchronization differences. This has been confirmed through manual inspection where the reported differences are compared with the ground truth. Since we have randomly labeled the original and changed programs as $P_1$ and $P_2$, some of the differences are in $\Delta_{12}$ whereas the others are reported in $\Delta_{21}$. In total, EC-Diff found 251 differences in $\Delta_{12}$ and 151 differences in $\Delta_{21}$.

The missed difference resides in r18169-3: after running the rank-3 analysis, our method still could not find it. The reason is because the differentiating behavior involves a deadlock and the patch that removed it. We explain why our method cannot detect it in Section 6.4.

6.3 Results on the Second Set of Benchmarks

Table 2 shows our results on the second set of benchmarks, consisting of six medium-sized programs. Note that these programs are already out of the reach of the prior technique [14] due to its requirement of manual code instrumentation; therefore, we only report the statistics of applying EC-Diff. Again, the original and modified programs are randomly labeled as $P_1$ and $P_2$, respectively, to facilitate evaluation.

In total EC-Diff found 30 differences in $\Delta_{12}$ and 42 differences in $\Delta_{21}$. Furthermore, all of them were found during rank-1 analysis, and confirmed by manual inspection. What is impressive is that these differences were identified by sifting through a combined
### Table 1: Experimental results on the first set of benchmark programs.

<table>
<thead>
<tr>
<th>Name</th>
<th>LoC</th>
<th>Threads</th>
<th>Type</th>
<th>Difference</th>
<th>Rank</th>
<th>EC-Diff</th>
<th>Prior Technique [14]</th>
<th>Time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2c-hid [14]</td>
<td>76</td>
<td>Sync.</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>1 12</td>
<td>1 12</td>
<td>2.570</td>
<td>2.570</td>
</tr>
<tr>
<td>i2c-hid-noa</td>
<td>70</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1,573</td>
<td>1,573</td>
</tr>
<tr>
<td>r8169-1 [14]</td>
<td>65</td>
<td>Order</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>870</td>
<td>852</td>
</tr>
<tr>
<td>r8169-2 [14]</td>
<td>80</td>
<td>Order</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>873</td>
<td>839</td>
</tr>
<tr>
<td>rtl8169-1</td>
<td>578</td>
<td>Order</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>1 0 3</td>
<td>1</td>
<td>60,741</td>
<td>60,691</td>
</tr>
<tr>
<td>rtl8169-2</td>
<td>578</td>
<td>Order</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>1 0 3</td>
<td>1</td>
<td>60,741</td>
<td>60,741</td>
</tr>
<tr>
<td>rtl8169-3</td>
<td>578</td>
<td>Order</td>
<td>yes</td>
<td>no</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>60,741</td>
<td>60,741</td>
</tr>
<tr>
<td>cherokee [52]</td>
<td>150</td>
<td>Sync.</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>0 2 1</td>
<td>1</td>
<td>1,148</td>
<td>1,148</td>
</tr>
<tr>
<td>transmission</td>
<td>91</td>
<td>Cond.</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>1 0 1</td>
<td>1</td>
<td>690</td>
<td>613</td>
</tr>
<tr>
<td>apache-21287</td>
<td>74</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1,406</td>
<td>1,406</td>
</tr>
<tr>
<td>apache-25220</td>
<td>181</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>8 0 3</td>
<td>3</td>
<td>3,206</td>
<td>3,206</td>
</tr>
<tr>
<td>account</td>
<td>82</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>1 1 0</td>
<td>1</td>
<td>7,289</td>
<td>6,655</td>
</tr>
<tr>
<td>barrier</td>
<td>134</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5,248</td>
<td>6,321</td>
</tr>
<tr>
<td>lazy</td>
<td>111</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>0 1 2</td>
<td>1</td>
<td>3,409</td>
<td>3,549</td>
</tr>
<tr>
<td>reorder</td>
<td>170</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>3 0 1</td>
<td>1</td>
<td>9,493</td>
<td>8,737</td>
</tr>
<tr>
<td>threadRW</td>
<td>147</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>2 0 1</td>
<td>0</td>
<td>9,092</td>
<td>8,552</td>
</tr>
<tr>
<td>lineEq-21</td>
<td>90</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>0 8 1</td>
<td>1</td>
<td>2,905</td>
<td>2,905</td>
</tr>
<tr>
<td>lineEq-39</td>
<td>127</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>2 0 1</td>
<td>0</td>
<td>35,014</td>
<td>35,014</td>
</tr>
<tr>
<td>mozilla</td>
<td>84</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>0 1 0</td>
<td>1</td>
<td>473</td>
<td>565</td>
</tr>
<tr>
<td>mysql-3596</td>
<td>92</td>
<td>Cond.</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>1 0 1</td>
<td>1</td>
<td>773</td>
<td>733</td>
</tr>
<tr>
<td>mysql-644</td>
<td>110</td>
<td>Sync.</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
<td>0 2 1</td>
<td>1</td>
<td>1,343</td>
<td>1,434</td>
</tr>
<tr>
<td>counter-seq</td>
<td>47</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>0 2 0</td>
<td>1</td>
<td>1,135</td>
<td>1,135</td>
</tr>
<tr>
<td>me-queue</td>
<td>116</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>2 0 0</td>
<td>5</td>
<td>5,754</td>
<td>5,754</td>
</tr>
<tr>
<td>mysql5</td>
<td>59</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>0 0 1</td>
<td>1</td>
<td>1,283</td>
<td>1,283</td>
</tr>
<tr>
<td>freebsd-a</td>
<td>176</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>0 0 22</td>
<td>1</td>
<td>7,910</td>
<td>10,109</td>
</tr>
<tr>
<td>libvm-9441</td>
<td>127</td>
<td>Cond.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>0 10 0</td>
<td>2</td>
<td>3,042</td>
<td>3,118</td>
</tr>
<tr>
<td>gcc-25530</td>
<td>87</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>0 0 1</td>
<td>1</td>
<td>806</td>
<td>806</td>
</tr>
<tr>
<td>gcc-3594</td>
<td>83</td>
<td>Sync.</td>
<td>yes</td>
<td>yes</td>
<td>2</td>
<td>0 2 1</td>
<td>1</td>
<td>1,843</td>
<td>1,843</td>
</tr>
<tr>
<td>gcc-21334</td>
<td>136</td>
<td>Sync.</td>
<td>yes</td>
<td>yes</td>
<td>2</td>
<td>8 0 0</td>
<td>2</td>
<td>5,290</td>
<td>5,290</td>
</tr>
<tr>
<td>gcc-40518</td>
<td>102</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>0 8 2</td>
<td>1</td>
<td>3,027</td>
<td>3,027</td>
</tr>
<tr>
<td>ghc-512624</td>
<td>95</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>1</td>
<td>1 180 0</td>
<td>1</td>
<td>5,748</td>
<td>5,748</td>
</tr>
<tr>
<td>jetty-1187</td>
<td>69</td>
<td>Sync.</td>
<td>yes</td>
<td>no</td>
<td>2</td>
<td>0 2 0</td>
<td>2</td>
<td>885</td>
<td>885</td>
</tr>
</tbody>
</table>

Total: 251 151 338,913 339,849 15.57 > 3h

∗ > 3600.00 means verification of the edge in P1 succeeded, but verification of the edge in P2 timed out after an hour.

---

6.4 Discussion

Now, we answer the two research questions.

Q1: Is EC-Diff accurate enough for identifying synchronization differences? The answer is yes. As shown in our experimental results, EC-Diff produced a large number of differences, the majority of which are at rank 1, which means they are individual read-from edges allowed in only one of the two programs, while the rest are at rank 2. Although we do not guarantee that EC-Diff finds all differentiating behaviors, these detected ones have been confirmed by manual inspection.

Given that these benchmarks contain real concurrency bug patterns reported and analyzed by many existing tools for testing and repair, the result of EC-Diff is sufficiently accurate. The success in a large part is due to the nature of these programs, where two versions behave almost same except for the thread synchronization. In such cases, our approximate analysis can come really close to the ground truth.

Q2: Is EC-Diff more efficient than the prior technique based on model checking? The answer is yes. As shown in our results, EC-Diff was 10x to 1000x faster and, in total, completed the differential analysis of 13,500 lines of multithreaded C code in about 160 seconds. In contrast, the prior technique took a much longer time to analyze these programs.
8 CONCLUSIONS

We have presented a fast and approximate static analysis method for computing the synchronization differences of two concurrent programs. The method uses Datalog to capture structural information of the programs, and uses a set of inference rules to codify the analysis algorithm. The analysis result, computed by an off-the-shelf Datalog solver, consists of sets of data-flow edges that are allowed by only one of the two programs. We implemented the proposed method and evaluated it on a large number of benchmark programs. Our results show the method is orders-of-magnitudes faster than the prior technique while being sufficiently accurate in identifying the actual differences.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation (NSF) under grant CCF-1727210, the Office of Naval Research (ONR) under grant N00014-17-1-2896, the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 678177).
REFERENCES