Datalog-Based Scalable Semantic Diffing of Concurrent Programs

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

Shuvendu K. Lahiri
Microsoft Research
Redmond, WA, USA

Constantin Enea
University Paris Diderot
Paris, France

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 our method 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

ACM Reference Format:
Chungha Sung, Shuvendu K. Lahiri, Constantin Enea, and Chao Wang. 2018. Datalog-Based Scalable Semantic Diffing of Concurrent Programs. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238211

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238211

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 [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 [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 interleavings 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 [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 interleavings would not work due to the often exponential blowup in the number of interleavings.

Our method avoids the problem by being *approximate* in that it does not enumerate interleavings. This also means infeasible behaviors are sometimes included. However, our approximation is carefully designed to take into consideration the program semantics

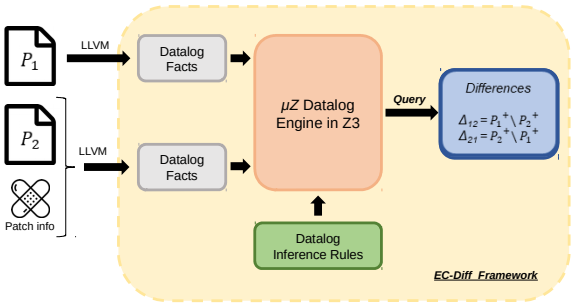


Figure 1: Overview of our semantic diffing method.

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 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 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 μZ [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.

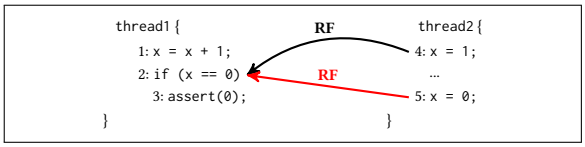
2 MOTIVATION

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.

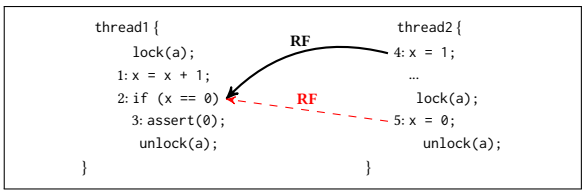
2.1 The First Example

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 `thread1` executes the statement at Line 2 right after `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 `unlock(a)` must be executed before the `lock(a)` in `thread1`, which in turn must be executed before Line 1 and Line 2.



(a) Before change



(b) After change

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

Since the store of x at Line 1 is the most recent, the load of x at Line 2 will get its value, not the value written at Line 5.

Thus, the allowed data-flow edges are as follows: RF(L4,L2) and RF(L5,L2) for the original program, and RF(L4,L2) 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.

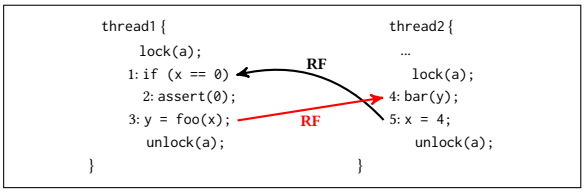
In addition to RF edges, there are other types of relations considered during our analysis, including program order, inter-thread order imposed by thread *create*, *join*, *signal-wait* as well as *store-store* order. Nevertheless, when interpreting the final results, we focus on differences in the RF 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

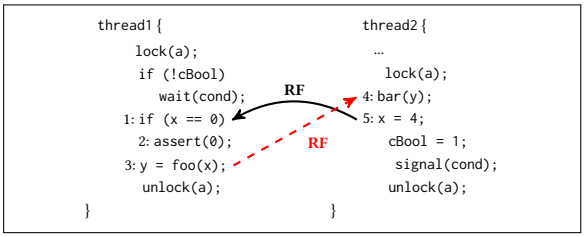
Fig. 3 shows a more sophisticated example: the use of *signal-wait*, which is often difficult for static analyzers. Since the variable x is initialized to 0, when the critical section in thread1 is executed before thread2, the load of x at Line 1 will get the value 0, which leads to the assertion violation 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



(a) Before change



(b) After change

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

Fig 2(a)	*mustHB*	{(1, 2), (2, 3), (1, 3), (4, 5)}
	mayHB	*mustHB* ∪ {(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5), (4, 1), (4, 2), ...}
	MayRF	{(4, 1), (4, 2), (5, 1), **(5,2)**}
Fig 2(b)	*mustHB*	{(1, 2), (2, 3), (1, 3), (4, 5)}
	mayHB	*mustHB* ∪ {(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5), (4, 1), (4, 2), ...}
	MayRF	{(4, 1), (4, 2), (5, 1)}

Fig 3(a)	*mustHB*	{(1, 2), (2, 3), (1, 3), (4, 5)}
	mayHB	*mustHB* ∪ {(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5), (4, 1), (4, 2), ...}
	MayRF	{**(3,4)**, (5, 1), (5, 3)}
Fig 3(b)	*mustHB*	{(1, 2), (2, 3), (1, 3), (4, 5), **(4,1),(4,2),(4,3),(5,1),(5,2),(5,3)**}
	mayHB	*mustHB*
	MayRF	{(5, 1), (5, 3)}

Figure 4: Analysis steps for programs in Figs. 3(a) and 3(b).

signal, and *signal* must happen before *wait*, which resides before Lines 1-3 in *thread1*. Thus, there is a cycle (contradiction).

2.3 How Our Method Works

Our method differs from prior techniques which rely on either enumerating interleavings and conducing pairwise comparison [44], or model checking based techniques [14]. Both are computationally expensive. Instead, we use lightweight static analysis.

Our method represents the control and data dependencies of each program as a set of Datalog *facts*. We also design a set of Datalog *inference rules*, which capture our algorithm for deriving new facts from existing facts. Leveraging a Datalog solver, we can repeatedly apply the inference rules over the facts until a fixed point is reached. We will explain details of our Datalog facts and inference rules in Section 4.

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 data-flow 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 P_1 but not in P_2. 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 $\mathbb{G}$ be the set of global variables shared by threads in P. For each $x \in \mathbb{G}$, let $W(x)$ denote a store instruction and $R(x)$ denotes a load instruction. Let $\mathbb{I}$ be the set of all instructions in the program. Any binary relation over these instructions is a subset of $\mathbb{I} \times \mathbb{I}$.

For example, $\hat{so} \subseteq \mathbb{I} \times \mathbb{I}$ is a relation that orders the store instructions; $W_1(x) < W_2(x)$ means $W_1 \in \mathbb{I}$ is executed before $W_2 \in \mathbb{I}$. Thus, in Fig. 2(a), (L1,L4), (L4,L1), (L1,L5), (L5,L1), (L4,L5) belong to $\hat{so}$, but (L5,L4) does not belong to $\hat{so}$ because it is not consistent with the program order.

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

Edges in $\hat{ss}$ are from either $\hat{rf}$ or $\hat{so}$ – 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 $\hat{T} = \langle \hat{so}, \hat{rf}, \hat{sets}, k \rangle$, where $\hat{so} \subseteq \{W_1(x) \times W_2(x) \mid W_1 \in \mathbb{I}, W_2 \in \mathbb{I}$, and $W_1 < W_2$ in some execution trace$\}$, $\hat{rf} \subseteq \{W(x) \times R(x) \mid W \in \mathbb{I}$ and $R \in \mathbb{I}\}$, and $\hat{sets} \subseteq \{\hat{ss} \subseteq \hat{rf} \cup \hat{so} \mid |\hat{ss}| \leq k\}$.*

Given the abstract traces $\hat{T}_1$ and $\hat{T}_2$ of two programs P_1 and P_2, respectively, we define their difference as $\Delta = (\Delta_{12}, \Delta_{21})$, where $\Delta_{12} = \hat{T}_1 \setminus \hat{T}_2$ and $\Delta_{21} = \hat{T}_2 \setminus \hat{T}_1$. Next, we define what it means for $\hat{T}_1$ to be a refinement of $\hat{T}_2$, denoted $\hat{T}_1 \subseteq \hat{T}_2$.

Definition 2 (Abstract Trace Refinement). *Given two abstract traces $\hat{T}_1 = \langle \hat{so}_1, \hat{rf}_1, \hat{sets}_1, k \rangle$ and $\hat{T}_2 = \langle \hat{so}_2, \hat{rf}_2, \hat{sets}_2, k \rangle$, we say $\hat{T}_1$ is a refinement of $\hat{T}_2$, denoted $\hat{T}_1 \subseteq \hat{T}_2$, if and only if $\hat{so}_1 \subseteq \hat{so}_2$, $\hat{rf}_1 \subseteq \hat{rf}_2$, and $\hat{sets}_1 \subseteq \hat{sets}_2$.*

That is, when $\hat{T}_1 \subseteq \hat{T}_2$, the abstract behavior of P_1 is covered by that of P_2. And the difference $(\hat{T}_2 \setminus \hat{T}_1)$ is characterized by $\hat{so}_2 \setminus \hat{so}_1$, $\hat{rf}_2 \setminus \hat{rf}_1$, and $\hat{sets}_2 \setminus \hat{sets}_1$. Finally, if the abstract traces of P_1 and P_2 refine each other, we say they are *rank-k equivalent*.

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

$$\text{PO}(s_1, s_2), \text{PO}(s_1, s_3), \text{PO}(s_2, s_4), \text{PO}(s_3, s_4), \text{PO}(s_4, s_5).$$

Then, we write down the Datalog inference rules:

$$\text{HB}(a, b) \leftarrow \text{PO}(a, b)$$
$$\text{HB}(c, e) \leftarrow \text{HB}(c, d) \wedge \text{HB}(d, e)$$

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(s_1, s_5) indeed holds whereas HB(s_2, s_3) 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(s_1, th_1): Statement s_1 resides in Thread th_1
- Po(s_1, s_2): Statement s_1 is before s_2 in a thread
- Dom(s_1, s_2): Statement s_1 dominates s_2 in a thread
- PostDom(s_1, s_2): s_1 post-dominates s_2 in a thread
- ThrdCreate(th_1, s_1, th_2): Thread th_1 creates th_2 at s_1
- ThrdJoin(th_1, s_1, th_2): Thread th_1 joins back th_2 at s_1
- CondWait(s_1, v_1): s_1 waits for condition variable v_1
- CondSignal(s_1, v_1): s_1 sends condition variable v_1
- Load(s_1, v_1): Statement s_1 reads from variable v_1
- Store(s_1, v_1): Statement s_1 writes to variable v_1
- InCS(s_1, l_1): s_1 resides in a critical section guarded by lock(l_1)–unlock(l_1) pair
- SameCS(s_1, s_2, l_1): s_1 and s_2 are in the same critical section guarded by l_1
- DiffCS(s_1, s_2, l_1): s_1 and s_2 are in different critical sections guarded by l_1

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: MayHb(s_1, s_2) means s_1 may happen before s_2 in some execution, and MustHb(s_1, s_2) means s_1 happens before s_2 in all executions when both occur. Since the program order in each thread implies the execution order, we have the following rule:

$$\text{MustHb}(s_1, s_2) \leftarrow \text{Po}(s_1, s_2)$$

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

By definition MustHb implies MayHb, which means

$$\text{MayHb}(s_1, s_2) \leftarrow \text{MustHb}(s_1, s_2)$$

4.2 Rules for Inter-thread Dependency

When a parent thread th_1 creates a child thread th_2 at the statement s_1, e.g., by invoking `pthread_create`, any statement s_2 in the child thread must occur after s_1.

$$\text{MustHb}(s_1, s_2) \leftarrow \text{ThrdCreate}(th_1, s_1, th_2) \wedge \text{St}(s_2, th_2)$$

Similarly, when a parent thread th_1 joins back a child thread th_2 at s_1, any statement s_2 in th_2 must occur before s_1.

$$\text{MustHb}(s_2, s_1) \leftarrow \text{ThrdJoin}(th_1, s_1, th_2) \wedge \text{St}(s_2, th_1)$$

```
thread1() {
      a = 1;
      cond = true;

}
```

```
thread2() {
      while(!cond) {}
      x = a;

}
```

Figure 5: *Ad hoc* synchronization (`cond = false` initially).

4.3 Rules for Signal-Wait Dependency

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

$$\text{MustHb}(s_1, s_2) \leftarrow \text{CondSignal}(v_1, s_1) \wedge \text{CondWait}(v_1, s_2)$$

However, the rule needs to be used with caution. In practice, *wait(c)* is often wrapped in an if-condition as shown in Figure 3(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 MustHb edge from `cond=true` to `while(!cond)`. This is similar to adding 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 MustHb is transitive, we use the following rule to compute the transitive closure:

$$\text{MustHb}(s_1, s_3) \leftarrow \text{MustHb}(s_1, s_2) \wedge \text{MustHb}(s_2, s_3)$$

When instructions in concurrent threads are not ordered by MustHb, we assume they may occur in any order:

$$\text{MayHb}(s_1, s_2) \leftarrow \text{St}(s_1, th_1) \wedge \text{St}(s_2, th_2) \wedge \neg\text{MustHb}(s_2, s_1)$$

The MayHb relation is also transitive:

$$\text{MayHb}(s_1, s_3) \leftarrow \text{MayHb}(s_1, s_2) \wedge \text{MayHb}(s_2, s_3)$$

4.6 Lock-Enforced Critical Section

For critical sections based on *lock-unlock*, we introduce rules based on access patterns. First, we compute CoveredStore(s_1, v_1, l_1), meaning the store in s_1 is overwritten by a subsequent store in the same critical section. Consider `lk(a)` → W_1`(v)` → W_2`(v)` → `unlk(a)`, where W_1`(v)` is a covered store and thus not visible to reads in other critical sections protected by the same lock.

$$\text{CoveredStore}(s_1, v_1, l_1) \leftarrow \text{Store}(s_1, v_1) \wedge \text{Store}(s_2, v_1) \wedge \text{PostDom}(s_2, s_1) \wedge \text{SameCS}(s_1, s_2, l_1)$$

Similarly, CoveredLoad(s_2, v_1, l_1) means the load of v_1 in s_2 is covered and thus can only read from a preceding store in the same

Figure 6: Differences of abstract traces: Δ_{12} (left) and $\Delta 21$ (right).

critical section.

$$\text{CoveredLoad}(s_2, v_1, l_1) \leftarrow \text{Store}(s_1, v_1) \wedge \text{Load}(s_2, v_1) \\ \wedge \text{Dom}(s_1, s_2) \wedge \text{SameCS}(s_1, s_2, l_1)$$

Consider `lk(a)` → `W(v)` → `R(v)` → `unlk(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 $\text{NoRf}(s_1, s_2)$ which means the read-from edge between s_1 and s_2 is infeasible.

$$\text{NoRf}(s_1, s_2) \leftarrow \text{Store}(s_1, v_1) \wedge \text{Store}(s_3, v_1) \wedge \text{Load}(s_2, v_1) \\ \wedge \text{MustHb}(s_1, s_3) \wedge \text{MustHb}(s_3, s_2)$$

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

$$\text{NoRf}(s_1, s_2) \leftarrow \text{Store}(s_1, v_1) \wedge \text{Load}(s_2, v_1) \wedge \text{MayHb}(s_1, s_2) \\ \wedge \text{CoveredLoad}(s_2, v_1, l_1) \wedge \text{DiffCS}(s_1, s_2, l_1)$$

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.

$$\text{NoRf}(s_1, s_2) \leftarrow \text{Store}(s_1, v_1) \wedge \text{Load}(s_2, v_1) \wedge \text{MayHb}(s_1, s_2) \\ \wedge \text{CoveredStore}(s_1, v_1, l_1) \wedge \text{DiffCS}(s_1, s_2, l_1)$$

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

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

$$\text{MayRf}(s_1, s_2) \leftarrow \text{Store}(s_1, v_1) \wedge \text{Load}(s_2, v_1) \wedge \text{MayHb}(s_1, s_2) \\ \wedge \neg \text{NoRf}(s_1, s_2)$$

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 $\hat{T}_1$ and $\hat{T}_2$, are represented by the circles with dashed lines. Conceptually, the symmetric difference is computed based on $\Delta_{12} = \hat{T}_1 \setminus \hat{T}_2$ and $\Delta_{21} = \hat{T}_2 \setminus \hat{T}_1$, 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 DiffP1 and DiffP2 and rules for computing them:

$$\text{DiffP1}(s_1, s_2) \leftarrow \text{MayRf}(s_1, s_2, P_1) \wedge \neg \text{MayRf}(s_1, s_2, P_2)$$
$$\text{DiffP2}(s_1, s_2) \leftarrow \text{MayRf}(s_1, s_2, P_2) \wedge \neg \text{MayRf}(s_1, s_2, P_1)$$

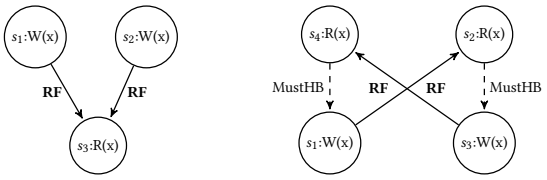


Figure 7: Illustrating the first two rank-2 inference rules.

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 MayHb edges (used to compute MayRf), and the number of MayHb 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 MayHb (and hence 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 MayRf relation, which was defined over two instructions (an edge), to MayRfs defined over four instructions, to represent an ordered set of (two) *read-from* edges. Similarly, we extend the NoRf relation to 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 $\text{MayRf}(s_1, s_2)$ means there may exist some execution trace that allows the *read-from* edge (s_1, s_2). Similarly, $\text{NoRfs}((s_1, s_2), (s_3, s_4))$ 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 $\text{MayRfs}((s_1, s_2), (s_3, s_4))$ 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 NoRfs, which in turn is used to compute MayRfs. Since it is not possible to enumerate all scenarios due to theoretical limitations, we resort to the most common scenarios. Nevertheless, we guarantee that NoRfs is an under-approximation, and the corresponding MayRfs is an over-approximation.

$$\text{NoRfs}((s_1, s_3), (s_2, s_3)) \leftarrow \text{MayRf}(s_1, s_3) \wedge \text{MayRf}(s_2, s_3)$$

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).

$$\text{NoRfs}((s_1, s_2), (s_3, s_4)) \leftarrow \text{MayRf}(s_1, s_2) \wedge \text{MayRf}(s_3, s_4) \\ \wedge \text{MustHb}(s_2, s_3) \wedge \text{MustHb}(s_4, s_1)$$

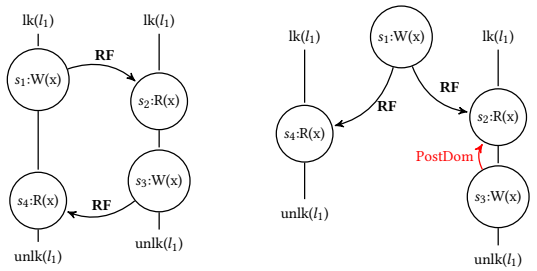


Figure 8: Illustrating rank-2 rules related to lock-unlock.

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.

$$\text{NoRFs}((s_1, s_2), (s_3, s_4)) \leftarrow \text{SameCS}(s_1, s_4, l_1) \wedge \text{SameCS}(s_2, s_3, l_1) \wedge \text{DiffCS}(s_1, s_2, l_1)$$

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_2, 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 $\text{PostDom}(s_3, s_2)$ to mean, after s_2 is executed, s_3 is guaranteed to be executed as well.

$$\text{NoRFs}((s_1, s_2), (s_1, s_4)) \leftarrow \text{Store}(s_3, v_1) \wedge \text{PostDom}(s_3, s_2) \wedge \text{DiffCS}(s_2, s_4, l_1) \wedge \text{SameCS}(s_2, s_3, l_1)$$

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_1, s_4) in the same execution trace, the store in s_3 contradicts to the read-from edge (s_1, s_4).

Finally, we compute MayRFs based on NoRFs:

$$\text{MayRFs}((s_1, s_2), (s_3, s_4)) \leftarrow \neg\text{NoRFs}((s_1, s_2), (s_3, s_4))$$

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 trace. With MayRFs, we compute differences (DiffP1 and 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.

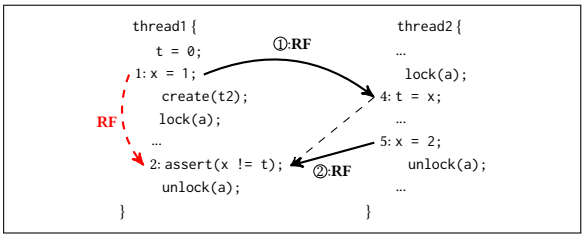
5.3 Example for Rank-2 Analysis

Fig. 9 shows an example that illustrates the rank-2 analysis. Here, `thread1` sets `t` to 0 and `x` to 1 before creating `thread2`. Due to lock-unlock pairs, the assertion cannot be violated in Fig. 9(a). However, if the lock-unlock in `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.

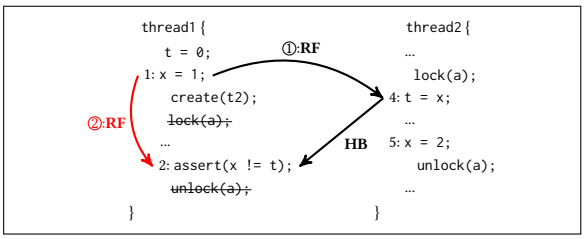
However, the synchronization difference cannot be captured by any individual `MayRF` edge. In fact, the table in Fig. 10 shows that the two programs have the same set of `MayRF` edges. In particular, since there are two stores of `x`, the load at Line 2 may read from both Line 1 and Line 5.

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

- Assume `RF(L1,L4)` occurs first, meaning `thread2` acquires the lock and thus prevents `thread1` from acquiring the same lock until `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, `RF(L1,L2)` cannot occur after `RF(L1,L4)` in the same execution.
- Assume `RF(L1,L2)` occurs first and `thread2` will not be executed until `thread1` finishes. In this case, `RF(L1,L4)` is allowed since no store of `x` is in `thread1`.



(a) Before change



(b) After change

Figure 9: Example programs with rank-2 differences.

Fig 9(a)	*mustHB*	$\{(1,2),(1,4),(1,5),(4,5)\}$
	mayHB	*mustHB* $\cup \{(2,4),(2,5),(4,2),(5,2)\}$
	MayRF	$\{(1,2),(1,4),(5,2)\}$
	Rank2	$\{[(1,2) \rightarrow (1,4)],[(1,4) \rightarrow (5,2)]\}$
Fig 9(b)	*mustHB*	$\{(1,2),(1,4),(1,5),(4,5)\}$
	mayHB	*mustHB* $\cup \{(2,4),(2,5),(4,2),(5,2)\}$
	MayRF	$\{(1,2),(1,4),(5,2)\}$
	Rank2	$\{[(1,2) \rightarrow (1,4)],[(1,4) \rightarrow (5,2)],$ $\mathbf{[(1,4) \rightarrow (1,2)]}\}$

Figure 10: Steps of our analysis for the programs in Fig. 9.

As a result, the program in Fig. 9(a) allows the ordered set {`RF(L1,L2)`, `RF(L1,L4)`} but not the ordered set {`RF(L1,L4)`, `RF(L1,L2)`}.

However, the program in Fig. 9(b) allows the ordered set {`RF(L1,L4)`, `RF(L1,L2)`} as well, due to the removal of the lock-unlock pairs in `thread1`. Specifically, when `RF(L1,L4)` occurs at the start of an execution, `thread1` may execute Line 2 before `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 μZ [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 μZ 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[1].

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.

[1] https://github.com/ChunghaSung/EC-Diff

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 $\Delta_{12} = \hat{T_1} \setminus \hat{T_2}$ and $\Delta_{21} = \hat{T_2} \setminus \hat{T_1}$. 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 MayHb edges (used to compute MayRf) in P_1 and P_2, respectively.

The last two columns compare the analysis time of our method and the model checking time of the prior technique [14] to check one different edge.

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 (*rtl8169-2*). 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 Δ_{12} whereas the others are reported in Δ_{21}. In total, EC-Diff found 251 differences in Δ_{12} and 151 differences in Δ_{21}.

The missed difference resides in *rtl8169-3*: 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 Δ_{12} and 42 differences in Δ_{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.

				EC-Diff							Prior Technique [14]
Name	LoC	Threads	Type	Difference	Rank	$\|\Delta_{12}\|$	$\|\Delta_{21}\|$	# of mayHB in P_1	# of mayHB in P_2	Time (s)	Time (s)
case1	52	3	Sync.	yes	1	0	7	1,343	1,343	0.26	11.53
case2	53	3	Cond.	yes	1	0	3	1,357	1,474	0.26	4.80
case3	67	3	Th.Order	yes	1	2	0	546	482	0.19	46.64
case4	94	3	Sync.	yes	2	0	1	421	421	0.20	8.59
i2c-hid [14]	76	3	Sync.	yes	1	1	0	2,570	2,570	0.28	27.28
i2c-hid-noa [14]	70	3	Sync.	yes	1	1	0	1,573	1,573	0.26	7.48
r8169-1 [14]	65	3	Order	yes	1	1	0	870	852	0.25	3.38
r8169-2 [14]	80	3	Order	yes	1	1	0	873	839	0.25	2.17
r8169-3 [14]	105	4	Order	yes	1	1	0	769	769	0.25	8.37
rtl8169-1 [14]	578	8	Order	yes	1	1	0	60,741	60,691	0.89	1580.16
rtl8169-2 [14]	578	8	Order	yes	1	1	0	60,741	60,741	0.89	2384.14
rtl8169-3 [14]	578	8	Order	no	3	0	0	60,741	60,741	2.40	0.00
cherokee [52]	150	3	Sync.	yes	1	0	2	1,148	1,148	0.31	7.59
transmission [52]	91	3	Cond.	yes	1	1	0	690	613	0.29	6.89
apache-21287 [52]	74	3	Sync.	yes	1	2	0	1,406	1,406	0.27	6.29
apache-25520 [52]	181	3	Sync.	yes	2	8	0	3,206	3,206	0.33	23.81
account [11]	82	4	Cond.	yes	1	0	2	3,701	3,881	0.30	13.46
barrier [11]	138	4	Cond.	yes	1	3	0	7,289	6,655	0.26	150.54
boop [11]	134	3	Sync.	yes	1	3	0	2,625	2,625	0.25	8.90
fibbench [11]	63	3	Cond.	yes	1	0	71	5,248	6,321	0.28	1483.33
lazy [11]	76	4	Cond.	yes	2	0	6	3,409	3,549	0.24	32.16
reorder [11]	170	5	Cond.	yes	1	3	0	9,493	8,737	0.40	12.79
threadRW [11]	147	5	Cond.	yes	1	2	0	9,092	8,552	0.30	7.57
lineEq-2t [13]	90	3	Sync.	yes	2	0	8	2,905	2,905	0.30	23.34
linux-iio [13]	114	3	Sync.	yes	1	3	0	5,851	5,851	0.31	24.13
linux-tg3 [13]	130	3	Cond.	yes	1	2	0	15,979	15,160	0.63	617.01
vectPrime [13]	127	3	Sync.	yes	2	2	0	35,014	35,014	0.52	2.22
mozilla-61369 [38]	84	3	Cond.	yes	1	0	1	473	565	0.25	3.57
mysql-3596 [38]	92	3	Cond.	yes	1	1	0	773	733	0.25	3.82
mysql-644 [38]	110	3	Cond.	yes	1	0	2	1,343	1,434	0.33	5.40
counter-seq [23]	47	3	Sync.	yes	2	0	2	1,135	1,135	0.26	18.13
ms-queue [23]	116	3	Sync.	yes	2	2	0	5,754	5,754	0.59	29.01
mysql5 [29]	59	3	Sync.	yes	2	0	4	1,283	1,283	0.20	22.92
freebsd-a [51]	176	4	Cond.	yes	1	0	22	7,910	10,109	0.33	25.40
llvm-8441 [7]	127	3	Cond.	yes	1	0	10	3,042	3,118	0.41	16.36
gcc-25530 [2]	87	3	Sync.	yes	2	2	0	806	806	0.20	12.15
gcc-3584 [3]	83	3	Sync.	yes	2	2	0	1,843	1,843	0.24	17.23
gcc-21334 [1]	136	3	Sync.	yes	2	8	0	5,290	5,290	0.35	195.20
gcc-40518 [4]	102	3	Sync.	yes	1	0	8	3,027	3,027	0.25	14.31
glib-512624 [5]	95	3	Sync.	yes	1	198	0	5,748	5,748	0.32	$*$ >3600.00
jetty-1187 [6]	69	3	Sync.	yes	2	0	2	885	885	0.22	19.34
Total						251	151	338,913	339,849	15.57	> 3h

$*$ > 3600.00 means verification of the edge in P_1 succeeded, but verification of the edge in P_2 timed out after an hour.

total of 24 million MayHb edges, and yet, the analysis of all programs took only 140 seconds. The efficiency is, in large part, due to the restriction of our analysis on instructions that access global variables as opposed to all instructions in the program (refer to the last paragraph of Section 5.1). Otherwise, the number of MayHb edges would have been orders-of-magnitude larger.

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.

Table 2: Experimental results on the second set of benchmark programs.

				EC-Diff						
Name	LoC	Threads	Type	Difference	Rank	$\|\Delta_{12}\|$	$\|\Delta_{21}\|$	# of mayHB in P_1	# of mayHB in P_2	Time (s)
pbzip-1 [49, 52]	1,143	5	Th.Order	yes	1	6	0	782,846	773,934	14.98
pbzip-2 [49, 52]	1,143	7	Th.Order	yes	1	12	0	1,150,404	1,135,428	30.61
aget-1 [49, 52]	1,523	4	Cond.	yes	1	4	0	1,099,047	1,078,695	9.41
aget-2 [49, 52]	1,523	6	Cond.	yes	1	8	0	3,218,034	3,162,684	28.60
pfscan-1 [49]	1,327	3	Cond.	yes	1	0	6	2,094,446	2,107,760	19.72
pfscan-2 [49]	1,327	5	Cond.	yes	1	0	36	4,138,361	4,164,989	39.96
Total						30	42	12,483,138	1,242,3490	140.28

```
thread1() {
    lock(a);
    lock(b);
    ...
    unlock(b);
    unlock(a);
}
```

```
thread1() {
    lock(b);
    lock(a);
    ...
    unlock(a);
    unlock(b);
}
```

Figure 11: Code from *rtl8169-3*: the original (left) and changed (right) versions.

Thus, we conclude that EC-Diff is effective in identifying synchronization differences in evolving programs. In practice, when developers update a program to fix concurrency bugs or remove performance bugs (e.g., by eliminating redundant locks), the differences in behavior are often reflected in (sets of) data-flow edges being feasible in one version but not in the other version. Thus, computing these (sets of) data-flow edges can be a fast way of checking if the changes introduce unexpected behaviors.

The Missing Case: Although EC-Diff detected most of the actual differences, it missed one in *rtl8169-3*. Fig. 11 shows the code snippet of `thread1` from the original program (P_1 on the left) and the changed program (P_2 on the right). The purpose of this patch is to resolve a deadlock issue by changing the acquisition order of locks. Since EC-Diff focuses solely on data-flow edges, it is not able to detect behavioral differences related to locking only. In some sense, this is a limitation shared by techniques relying on the notion of abstract traces [14, 44]: the two programs do not have data-related semantic difference other than the fact that a deadlock exists in one program but does not exist in the other program.

7 RELATED WORK

There has been prior work on statically computing the semantic differences of sequential and concurrent programs.

For sequential programs, Jackson and Ladd [26] proposed a method for computing the semantic differences by summarizing and comparing the dependencies between input and output. Godline and Strichman [18] proposed the use of inference rules to prove the equivalence of two programs. In the *SymDiff* project, Lahiri et al. [33, 34] developed a language-agnostic assertion checking tool for computing the differences of imperative programs. In the context of incremental symbolic execution [42], various change-impact analysis techniques were used to identify instructions that are affected by code modification and use the information to compute the corresponding test inputs [39]. However, these methods are not directly applicable to concurrent programs.

For concurrent programs, Joshi et al. [28] proposed the use of failure frequencies of assertions to compare two programs, while the general framework of refinement checking [8] could also be applied to traces of two programs. However, these techniques are limited to individual executions. Change-impact analysis [36] were also applied to concurrent programs, e.g., in regression testing [54], prioritized scheduling [27, 30], and incremental symbolic execution [19, 53]. However, these techniques focus on reducing the cost of testing and analysis as opposed to identifying the synchronization differences.

As we have mentioned earlier, the most closely related work is that of Bouajjani et al. [14], which computes the differences between partial data-flow dependencies of two concurrent programs using a bounded model checker. However, the method is costly; furthermore, it requires code instrumentation to insert assertions so they can be verified using a model checker. For example, it took about 30 minutes for a program (rtl8169) that can be analyzed by our method in less than a second.

Our method relies on the Datalog-based declarative program analysis framework, which previously has been applied to both sequential and concurrent programs as well as web applications [10, 15, 17, 19–22, 25, 35, 37, 41, 45, 48, 56]. In the context of static analysis of concurrent programs, for example, Kusano and Wang [31, 32] used Datalog in a thread-modular abstract interpretation to check the feasibility of inter-thread data-flow edges on sequentially consistent and weaker memory models. Sung et al. [46] used a similar technique for modeling preemption scheduling of interrupts and thus improving the accuracy of static analysis for interrupt-driven programs. However, none of these existing methods computes the synchronization differences of evolving 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-1722710, 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

[1] Gcc bug 21334. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=21334.
[2] Gcc bug 24430. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=25330.
[3] Gcc bug 3584. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=3584.
[4] Gcc bug 40518. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=40518.
[5] Glib bug 51264. https://bugzilla.gnome.org/show_bug.cgi?id=512624.
[6] Jetty bug 1187. https://jira.codejaus.org/browse/JETTY-1187.
[7] Llvm bug 8441. http://llvm.org/bugs/show_bug.cgi?id=8441.
[8] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
[9] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *ACM/IEEE international symposium on Microarchitecture*, Dec 2003.
[10] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 689–706, 2017.
[11] Dirk Beyer. Software verification and verifiable witnesses. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 401–416, 2015.
[12] Sandeep Bindal, Sorav Bansal, and Akash Lal. Variable and thread bounding for systematic testing of multithreaded programs. In *International Symposium on Software Testing and Analysis*, pages 145–155, 2013.
[13] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Außerlechner, and Raphael Spörk. Synthesis of synchronization using uninterpreted functions. In *International Conference on Formal Methods in Computer-Aided Design*, pages 11:35–11:42, 2014.
[14] Ahmed Bouajjani, Constantin Enea, and Shuvendu K. Lahiri. *Abstract Semantic Diffing of Evolving Concurrent Programs*, pages 46–65. Springer International Publishing, Cham, 2017.
[15] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 243–262, 2009.
[16] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–126, 1996.
[17] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–308, 2012.
[18] Benny Godlin and Ofer Strichman. Time for verification. chapter Inference Rules for Proving the Equivalence of Recursive Procedures, pages 167–184. Springer-Verlag, Berlin, Heidelberg, 2010.
[19] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 531–542, 2016.
[20] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.
[21] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming*, pages 2–27, 2006.
[22] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.
[23] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
[24] Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462, 2011.
[25] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115, 1995.
[26] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *International Conference on Software Maintenance*, pages 243–252, 1994.
[27] Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Change-aware preemption prioritization. In *International Symposium on Software Testing and Analysis*, pages 133–143, 2011.
[28] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 19–30, 2012.
[29] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, pages 165–176, 2015.
[30] Markus Kusano and Chao Wang. Assertion guided abstraction: A cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.
[31] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 799–809, 2016.
[32] Markus J. Kusano and Chao Wang. Thread-modular static analysis for relaxed memory models. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2017.
[33] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*, pages 712–717, 2012.
[34] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 345–355, 2013.
[35] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–12, 2005.
[36] Steffen Lehnert. A taxonomy for software change impact analysis. In *International Workshop on Principles of Software Evolution and Annual ERCIM Workshop on Software Evolution*, pages 41–50, 2011.
[37] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, pages 18–18, 2005.
[38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.
[39] Paul Dan Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 235–245, 2013.
[40] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.
[41] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.
[42] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 226–237, 2008.
[43] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.
[44] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
[45] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static DOM event dependency analysis for testing web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 447–459, 2016.
[46] Chungha Sung, Markus Kusano, and Chao Wang. Modular verification of interrupt-driven software. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 206–216, 2017.
[47] Chao Wang, Yu Yang, Aarti Gupta, and Ganesh Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.
[48] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.
[49] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. Technical report, University of Utah, 2008.
[50] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Chao Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *International SPIN Workshop on Model Checking Software*, pages 279–295, 2009.
[51] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 26–36, 2011.
[52] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *International Symposium on Computer Architecture*, pages 325–336, 2009.
[53] Tingting Yu, Zunchen Huang, and Chao Wang. ConTesa: Directed test suite augmentation for concurrent software. *IEEE Transactions on Software Engineering*, 2018.
[54] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimRT: An automated framework to support regression testing for data races. In *International Conference on Software Engineering*, pages 48–59, 2014.
[55] Tingting Yu, Tarannum S. Zaman, and Chao Wang. DESCRY: reproducing system-level concurrency failures. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 694–704, 2017.
[56] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 239–248, 2014.