Regression-free Synthesis for Concurrency[*]

Pavol Černý[1], Thomas A. Henzinger[2], Arjun Radhakrishna[2], Leonid Ryzhyk[34], and Thorsten Tarrach[2]

[1] University of Colorado Boulder
[2] IST Austria
[3] University of Toronto
[4] NICTA, Sydney, Australia[*]

Abstract. While fixing concurrency bugs, program repair algorithms may introduce new concurrency bugs. We present an algorithm that avoids such regressions. The solution space is given by a set of program transformations we consider in the repair process. These include reordering of instructions within a thread and inserting atomic sections. The new algorithm learns a constraint on the space of candidate solutions, from both positive examples (error-free traces) and counterexamples (error traces). From each counterexample, the algorithm learns a constraint necessary to remove the errors. From each positive examples, it learns a constraint that is necessary in order to prevent the repair from turning the trace into an error trace. We implemented the algorithm and evaluated it on simplified Linux device drivers with known bugs.

1 Introduction

The goal of program synthesis is to simplify the programming task by letting the programmer specify (parts of) her intent declaratively. *Program repair* is the instance of synthesis where we are given both a program and a specification. The specification classifies the execution of the program into *good traces* and *bad traces*. The synthesis task is to automatically modify the program so that the bad traces are removed, while (many of) the good traces are preserved.

In *program repair for concurrency*, we assume that all errors are caused by concurrent execution. We formalize this assumption into a requirement that all preemption-free traces are good. The program may contain concurrency errors that are triggered by more aggressive, preemptive scheduling. Such errors are notoriously difficult to detect and, in extreme cases, may only show up after years of operation of the system. Program repair for concurrency allows the programmer to focus on the preemption-free correctness, while putting the intricate task of proofing the code for concurrency to the synthesis tool.

Program repair for concurrency. The specification is provided by assertions placed by the programmer in the code. A trace, which runs without any assertion failure, is called "good", and conversely a trace with an assertion failure is "bad". We assume that the good traces specify the intent of the programmer. A trace is complete if every thread finishes its execution. A trace of a multi-threaded program is preemption-free if a thread is de-scheduled only at preemption-points, i.e., when a thread tries to execute a blocking operation, such as obtaining a lock.

Given a multithreaded program in which all complete preemption-free traces are good, the program repair for concurrency problem is to find a program for which the following two conditions hold: (a) all bad traces of the original program are removed; and (b) all the complete preemption-free traces are preserved. We further extend this problem statement by saying that if not all preemption-free traces are good, but all complete sequential traces are good, then we need to find a program such that (a) holds, and all complete sequential traces are preserved.

Regression-free algorithms. Let us consider a trace-based algorithm for program repair, that is, an iterative algorithm that in each iteration is given a trace (good or bad) of the program-under-repair, and produces a new program based on the traces seen. We say that such an algorithm is *regression-free* if after every iteration, we have that: first, all bad traces examined so far are removed, and second, all good traces examined so far are not turned into bad traces of the new program. (Of course, to make this definition precise, we will need to define a correspondence between traces of the original program and the new program.)

Program transformations. In order to remove bad traces, we apply the following program transformations: (1) reordering of adjacent instructions $i_1; i_2$ within a thread if the instructions are sequentially independent (i.e., if $i_1; i_2$ is sequentially equivalent to $i_2; i_1$), and (2) inserting atomic sections. The reordering of instructions is given priority as it may result in a better performance than the insertion of atomic sections. Furthermore, the reordering of instructions removes a surprisingly large number of concurrency bugs that occur in practice; according to a study of how programmers fix concurrency bugs in Linux device drivers [4], reordering of instructions is the most commonly used.

Our algorithm. Our algorithm learns constraints on the space of candidate solutions from both good traces and bad traces. We explain the constraint learning using as an example the program transformation (1), which reorders instructions within threads. From a bad trace, we learn reordering constraints that eliminate the counterexample using the algorithm of [4]. While eliminating the counterexample, such reorderings may transform a (not necessarily preemption-free) good trace into a bad trace — this would constitute a regression. In order to avoid regressions, our algorithm learns also from good traces. Intuitively, from a good trace π, we want to learn all the ways in which π can be transformed by reordering without turning it into an error trace— this is expressed as a program constraint. The program constraint is (a) sound, if all programs satisfying the constraint are regression-free; and (b) complete, if all programs violating the constraint have regressions. However, as learning a sound and complete constraint is computationally expensive, given a good trace π we learn a sound constraint

that only guarantees that π is not transformed into a bad trace. We generate the constraint using data-flow analysis on the instructions in π. The main idea of the analysis is that in good traces, the data-flow into passing assertions is protected by synchronization mechanisms (such as locks) and data-flow into conditionals along the trace. This protection may fail if we reorder instructions. We thus find a constraint that prevents such bad reorderings.

Summarizing, as the algorithm progresses and sees a set of bad traces and a set of good traces, it learns constraints that encode the ways in which the program can be transformed in order to eliminate the bad traces without turning the good traces into bad traces of the resulting program.

CEGIS vs PACES. A popular recent approach to synthesis is counterexample-guided inductive synthesis (CEGIS) [17]. Our algorithm can be viewed as an instance of CEGIS with the important feature that we learn from positive examples. We dub this approach PACES, for *Positive- and Counter-Examples in Synthesis*. The input to the CEGIS algorithm is a specification φ (possibly in multiple pieces – say, as a temporal formula and a language of possible solutions [3]). In the basic CEGIS loop, the synthesizer proposes a candidate solution S, which is then checked against φ. If it is correct, the CEGIS loop terminates; if not, a counterexample is provided and the synthesizer uses it to improve S. In practice, the CEGIS loop often faces performance issues, in particular, it can suffer from regressions: new candidate solutions may introduce errors that were not present in previous candidate solutions. We address this issue by *making use of positive examples* (good traces) in addition to counterexamples (bad traces). The good traces are used to learn constraints that ensure that these good traces are preserved in the candidate solution programs proposed by the CEGIS loop. The PACES approach applies in many program synthesis contexts, but in this paper, we focus on program repair for concurrency.

Related work. The closest related work is by von Essen and Jobstmann [7], which continues the work on program repair [11,9,12]. In [7], the goal is to repair reactive systems (given as automata) according to an LTL specification, with a guarantee that good traces do not disappear as a result of the repair. Their algorithm is based on the classic synthesis algorithm which translates the LTL specification to an automaton. In contrast, we focus on the repair of concurrent programs, and our algorithm uses positive examples and counterexamples.

There are several recent algorithms for inserting synchronization by locks, fences, atomic sections, and other synchronization primitives ([18,5,6,16]). Deshmukh et al. [6] is the only one of these which uses information about the correct parts of the program in bug fixing – a proof of sequential correctness is used to identify positions for locks in a concurrent library that is sequentially correct. CFix (Jin et al. [10]) can detect and fix concurrency bugs using specific bug detection patterns and a fixing strategy for each pattern of bug. Our approach relies on a general-purpose model checker and does not use any patterns.

Our algorithm for fixing bad traces starts by generalizing counterexample traces. In verification (as opposed to synthesis), concurrent trace generalization was used by Sinha et al. [14,15]; and by Alglave et al. [2] for detecting errors due

to weak memory models. Generalizations of good traces was previously used by Farzan et al. [8], who create an inductive data-flow graph (iDFG) to represent a proof of program correctness. They do not attempt to use iDFGs in synthesis.

We use the model checker CBMC [1] to generate both good and bad traces. Sen introduced concurrent directed random testing [13], which can be used to obtain good or bad traces much faster than a model checker. For a 30k LOC program their tool needs only about 2 seconds. We could use this tool to initially obtain good and bad traces faster, thus increasing the scalability of our tool.

Illustrative example. We motivate our approach on the program P in Figure 1a. There is a bug witnessed by the following trace: $\pi_1 = A \to B \to 1 \to 2 \to 3$ (the assertion at line 3 fails). Let us attempt to fix the bug using the algorithm from [4]. The algorithm discovers possible fixes by first generalizing the trace into a partial order (Figure 1b, without the dotted edges) representing the happens-before relations necessary for the bug to occur, and second, trying to create a cycle in the partial order to eliminate the generalized counterexample. It finds three possible ways to do this: swapping B and C, or moving C before A, or moving A after C, indicated by the dotted edges in Figure 1b. Assume that we continue with swapping B and C to obtain program P_1 where the first thread is $A; C; B$. Program P_1 contains an error trace $\pi_2 = A \to C \to n \to p$ (the assertion at line p fails). This bug was not in the original program, but was introduced by our fix. We refer to this type of bug as a regression.

In order to prevent regressions, the algorithm learns from good traces. Consider the following good trace $\pi_3 = A \to B \to C \to 1 \to 2 \to n \to 3 \to p$. The algorithm analyses the trace, and produces the graph in Figure 1c. Here, the thick red edges indicate the reads-from relation for `assert` commands, and the dashed blue edges indicate the reads-from relation for `await` commands. Intuitively, the algorithm now analyses why the assertion at line p holds in the given trace. This assertion reads the value written in line B (indicated by the thick red edge). The algorithm finds a path from B to p composed entirely from intra-thread sequential edges ($B \to C$ and $n \to p$) and dashed blue edges ($C \to n$). This path guarantees that this trace cannot be changed by different scheduler choices into a path where p reads from elsewhere and fails. From the good trace π_2 we thus find that there could be a regression unless B precedes C and n precedes p. Having learned this constraint, the synthesizer can find a better way to fix π_1. Of the three options described above, it chooses the only way which does not reorder B and C, i.e., it moves A after C. This fixes the program without regressions.

2 Programming Model and the Problem Statement

Our programs are composed of a fixed number (say n) threads written in the CWHILE language (Figure 2). Each statement has a unique program location and each thread has unique initial and final program locations. Further, we assume that execution does not stop on assertion failure, but instead, a variable *err* is set to 1. The `await` construct is a blocking assume, i.e., execution of

```
init: x = 0; y = 0; z = 0
thread1          thread2 thread3
1: await(x==1)  A: x:=1 n: await(z==1)
2: await(y==1)  B: y:=1 p: assert(y==1)
3: assert(z==1) C: z:=1
```

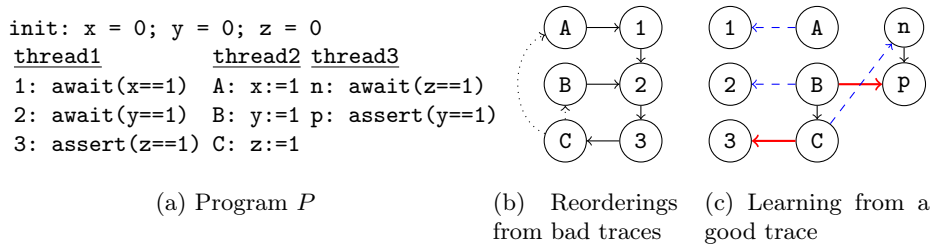(a) Program P (b) Reorderings from bad traces   (c) Learning from a good trace

Fig. 1: Program analysis with good and bad traces

`await(cond)` stops till `cond` holds. For example, a lock construct can be modelled as `atomic { await(lock_var == 0); lock_var := 1 }`. Note that `await` is the only blocking operation in CWHILE – hence, we call the `await` operations *preemption-points*.

```
iexp ::= iexp + iexp | iexp / iexp | iexp * iexp | var | constant
bexp ::= iexp >= iexp | iexp == iexp | bexp && bexp | !bexp
stmt ::= variable := iexp | variable := bexp | stmt; stmt | assume(bexp)
         | if (*) stmt else stmt | while (*) stmt | atomic { stmt }
         | assert(bexp) | await(bexp)
thrd ::= stmt                              prog  ::= thrd | prog‖thrd
```

Fig. 2: Syntax of programming language

Semantics. The *program-state* S of a program P is given by $(\mathcal{D}, (l^1, \ldots, l^n))$ where $\mathcal{D}$ is a valuation of variables, and each l^t is a thread t program location. Execution of the thread t statement at location l^t is represented as Sl^tS' where $S = (\mathcal{D}, (\ldots, l^t, \ldots))$ and $S' = (\mathcal{D}', (\ldots, l^{t'}, \ldots))$, and $l^{t'}$ and $\mathcal{D}'$ are the program location and variable valuation after executing the statement from $\mathcal{D}$. A *trace* π of P is a sequence $S_0l_0 \ldots S_m$ where (a) $S_0 = (\mathcal{D}, (l_\iota^1, \ldots, l_\iota^n))$ where each l_ι^t is the initial location of thread t; and (b) each $S_il_iS_{i+1}$ is a thread t transition for some t. Trace π is *complete* if $S_m = (\mathcal{D}_m, (l_f^1, \ldots, l_f^k))$, where each l_f^t is the final location of thread t. We say $S_il_i \ldots S_n$ is *equal modulo error-flag* to $S_i'l_i \ldots S_n'$ if each S_k and S_k' differ only in the valuation of the variable *err*.

Trace π is *preemption-free* if every context-switch occurs either at a preemption-point (`await` statement) or at the end of a thread's execution, i.e., if where $S_il_iS_{i+1}$ and $S_{i+1}l_{i+1}S_{i+2}$ are transitions of different threads (say threads t and t'), either the next thread t instruction after l_i is an `await`, or the thread t is in the final location in S_{i+1}. Similarly, we call a trace *sequential* if every context-switch happens at the end of a thread's execution.

A trace $\pi = S_0l_0 \ldots S_m$ is *bad* if the error variable *err* has values 0 and 1 in S_0 and S_m, respectively; otherwise, π is *good* trace. We assume that the bugs present in the input programs are *data-independent* – if $\pi = S_0l_0S_1 \ldots S_n$ is bad, so is every trace $\pi' = S_0'l_0'S_1' \ldots S_n'$ where $l_i = l_i'$ for all $0 \le i < n$.

Program transformations and Program constraints. We consider two kinds of transformations for fixing bugs:

- A *reordering transformation* $\theta = l_1 \leftrightsquigarrow l_2$ transforms P to P' if location l_1 immediately precedes l_2 in P and l_2 immediately precedes l_1 in P'. We only consider cases where the sequential semantics are preserved, i.e., if (a) l_1 and l_2 are from the same basic block; and (b) $l_1; l_2$ is equivalent to $l_2; l_1$.

– An *atomic section transformation* $\theta = [l_1; l_2]$ transforms P to P' if neighbouring locations l_1 and l_2 are in an atomic section in P', but not in P.

We write $P \xrightarrow{\theta_1 \ldots \theta_k} P'$ if applying each of θ_i in order transforms P to P'. We say transformation θ *acts across preemption-points* if either $\theta = l_1 \leftrightsquigarrow l_2$ and one of l_1 or l_2 is a preemption-point; or if $\theta = [l_1; l_2]$ and l_2 is a preemption-point.

Given a program P, we define *program constraints* to represent sets of programs that can be obtained through applying program transformations on P.

– *Atomicity constraint*: Program $P' \models [l_i; l_j]$ if l_i and l_j are in an atomic block.
– *Ordering constraint*: Program $P' \models l_i \leq l_j$ if l_i and l_j are from the same basic block and either l_i occurs before l_j, or P' satisfies $[l_i; l_j]$.

If $P' \models \Phi$, we say that P' *satisfies* Φ. Further, we define conjunction of Φ_1 and Φ_2 by letting $P' \models \Phi_1 \wedge \Phi_2 \Leftrightarrow (P' \models \Phi_1 \wedge P' \models \Phi_2)$.

Trace Transformations and Regressions. A trace $\pi = S_0 l_0 \ldots S_m$ *transforms* into a trace $\pi' = S'_0 l'_0 \ldots S'_m$ by *switching* if: (a) $S_0 l_0 \ldots S_n = S'_0 l'_0 \ldots S'_n$ and the suffixes $S_{n+2} l_{n+2} \ldots S_m$ and $S'_{n+2} l'_{n+2} \ldots S'_m$ are equal modulo error-flag; and (b) $l_n = l'_{n+1} \wedge l_{n+1} = l'_n$. We label switching transformations as a:

– *Free transformation* if l_n and l_{n+1} are from different threads. We write $\pi' \in f(\pi)$ if a sequence of free transformations takes π to π'.
– *Reordering transformation* $\theta = l^\sharp \leftrightsquigarrow l^\flat$ *acting on* π if $l_n = l^\sharp$ and $l_{n+1} = l^\flat$. We have $\pi' \in \theta(\pi)$ if repeated applications of θ transformations acting on π give π'. Similarly, $\pi' \in \theta^f(\pi)$ if repeated applications of θ and free transformations acting on π give π'.

Similarly, π' is obtained by *atomicity transformation* $\theta = [l_1, l_2]$ *acting on a trace* π if $\pi' \in f(\pi)$, and there are no context-switches between l_1 and l_2 in π'.

Trace analysis graphs. We use trace analysis graphs to characterize data-flow and scheduling in a trace. First, given a trace $\pi = S_0 l_0 \ldots$, we define the function *depends* to recursively find the data-flow edges into the l_i. Formally, $depends(i) = \cup_v \{(last(i,v), i)\} \cup depends(last(i,v))$ where v ranges over variables read by l_i, and $last(i,v)$ returns j if l_i reads the value of v written by l_j and $last(i,v) = \bot$ if no such j exists. As the base case, we define $depends(\bot) = \emptyset$.

Now, a *trace analysis graph* for trace $\pi = S_0 l_0 \ldots S_n$ is a multi-graph $G(\pi) = \langle V, \rightarrow \rangle$, where $V = \{\bot\} \cup \{i | 0 \leq i \leq n\}$ are the positions in the trace along with $\bot$ (representing the initial state) and $\rightarrow$ contains the following types of edges.

1. *Intra-thread order* (*IntraThreadOrder*): We have $x \rightarrow y$ if either $x < y$, and l_x and l_y are from the same thread, or if $x = \bot$.
2. *Data-flow into conditionals* (*DFConds*): We have $\bigcup_{a \in conds} depends(a) \subseteq \rightarrow$ where $x \in conds$ iff l_x is an assume or an await statement.
3. *Data-flow into assertions* (*DFAsserts*): We have $\bigcup_{a \in asserts} depends(a) \subseteq \rightarrow$ where $x \in asserts$ iff l_x is an assert statement.
4. *Non-free order* (*NonFreeOrder*): We have $x \rightarrow y$ if l_x and l_y write two different values to the same variable. Intuitively, the non-free orders prevent switching transformations that switch l_x and l_y.

Regressions. Suppose $P \xrightarrow{\theta_1, \ldots, \theta_k} P'$. We say $\theta_1, \ldots, \theta_k$ introduces a *regression* with respect to a good trace $\pi = S_0 l_0 \ldots S_m$ of P if there exists a trace $\pi' =$

$S'_0 l'_0 \ldots S'_m \in \theta_k^f \circ \ldots \circ \theta_1^f(\pi)$ such that: (a) π' is a bad trace of P'; (b) π does not freely transform into any bad trace of P; and (c) for every data-flow into conditionals edge $x \to y$ (say l_y reads the variables $\mathcal{V}$ from l_x) in $G(\pi)$, the edge $p(x) \to p(y)$ is a data-flow into conditionals edge in $G(\pi')$ (where $l'_{p(y)}$ reads the same variables $\mathcal{V}$ from $l'_{p(x)}$). Here, $p(i)$ is the position in π' of instruction at position i in π after the sequence of switching transformations that take π to π'. We say $\theta_1 \ldots \theta_k$ introduces a regression with respect to a set T_G of good traces if it introduces a regression with respect to at least one trace $\pi \in T_G$.

Intuitively, a program-transformation induces a regression if it allows a good trace π to become a bad trace π' due to the program transformations. Further, we require that π and π' have the conditionals enabled in the same way, i.e., the `assume` and `await` statements read from the same locations.

Remark 1. The above definition of regression attempts to capture the intuition that a good trace transforms into a "similar" bad trace. The notion of similar asks that the traces have the same data-flow into conditionals – this condition can be relaxed to obtain more general notions of regression. However, this makes trace analysis and finding regression-free fixes much harder (See Example 3).

Example 1. In Figure 1, the trace $\pi = A; B; C; n; p$ transforms under $B \leftrightsquigarrow C$ to $\pi' = A; C; B; n; p$, which freely transforms to $\pi'' = A; C; n; p; B$. Hence, $B \leftrightsquigarrow C$ introduces a regression with respect to π as π does not freely transform into a bad trace, and π' is bad while the `await` in n still reads from C.

The Regression-free Program-Repair Problem. Intuitively, the program-repair problem asks for a correct program P' that is a transformation of P. Further, P' should preserve all sequential behaviour of P; and if all preemption-free behaviour of P is good, we require that P' preserves it.

Program repair problem. The input is a program P where all complete sequential traces are good. The result is a sequence of program transformations $\theta_1 \ldots \theta_n$ and P', such that (a) $P \xrightarrow{\theta_1 \ldots \theta_n} P'$; (b) P' has no bad traces; (c) for each complete sequential trace π of P, there exists a complete sequential trace π' of P' such that $\pi' \in \theta_1 \circ \theta_2 \ldots \circ \theta_n(\pi)$; and (d) if all complete preemption-free traces of P are good, then for each such trace π, there exists a complete preemption-free trace π' of P' such that $\pi' \in \theta_1 \circ \theta_2 \ldots \circ \theta_n(\pi)$. We call the conditions (c) and (d) the *preservation of sequential and correct preemption-free behaviour.*

Regression-free error fix. Our approach to the above problem is through repeated regression-free error fixing. Formally, the regression-free error fix problem takes a set of good traces T_G, a program P and a bad trace π as input, and produces transformations $\theta_1, \ldots, \theta_k$ and P' such that $P \xrightarrow{\theta_1 \ldots \theta_k} P'$, $\pi' \in \theta_k^f \circ \ldots \circ \theta_1^f(\pi)$ is a trace in P', and $\theta_1, \ldots, \theta_k$ does not introduce a *regression* with respect to T_G.

3 Good and Bad Traces

Our approach to program-repair is through learning regression preventing constraints from good traces and error eliminating constraints from bad traces.

3.1 Learning from Good Traces

Given a trace π of P, a program constraint Φ is a *sound regression preventing constraint* for π if every sequence of program transformations $\theta_1, \ldots, \theta_k$, such that $P \xrightarrow{\theta_1 \ldots \theta_k} P'$ and $P' \models \Phi$, does not introduce a regression with respect to π. Further, if every $\theta_1 \ldots \theta_k$, such that $P \xrightarrow{\theta_1 \ldots \theta_k} P'$ and $P' \not\models \Phi$, introduces a regression with respect to π, then Φ is a *complete regression preventing constraint*.

Example 2. Let the program P be $\{1 : \mathtt{x} := 1; 2 : \mathtt{y} := 1\} || \{\mathtt{A} : \mathtt{await}(\mathtt{y} = 1);$ $\mathtt{B} : \mathtt{assert}(\mathtt{x} = 1)\}$. In Figure 3a, the constraint $\Phi^* = (1 < 2 \wedge A < B)$ is a sound and complete regression-preventing constraint for the trace $1 \to 2 \to A \to B$.

Lemma 1. *For a program P and a good trace π, the sound and complete regression-preventing constraint Φ^* is computable in exponential time in $|\pi|$.*

Intuitively, the proof relies on an algorithm that iteratively applies all possible free and program transformations in different combinations (there are a finite, though exponential, number of these) to π. It then records the constraints satisfied by programs obtained by transformations that do not introduce regressions.

The sound and complete constraints are usually large and impractical to compute. Instead, we present an algorithm to compute sound regression-preventing constraints. The main issue here is non-locality, i.e., statements that are not close to the assertion may influence the regression-preventing constraint.

Example 3. The trace in Figures 3b is a simple extension of Figure 3a. However, the constraint $(1 \leq 2 \wedge A \leq B)$ (from Example 2) does not prevent regressions for Figure 3b. An additional constraint $B \leq C \wedge 3 \leq 4$ is needed as reordering these statements can lead to the assertion failing by reading the value of $\mathtt{x}$ "too late", i.e., from the statement 4 (trace: $1 \to 2 \to A \to C \to 3 \to 4 \to B$).

Figure 3c clarifies our definition of regression, which requires that the data-flow edges into assumptions and awaits need to be preserved. The await can
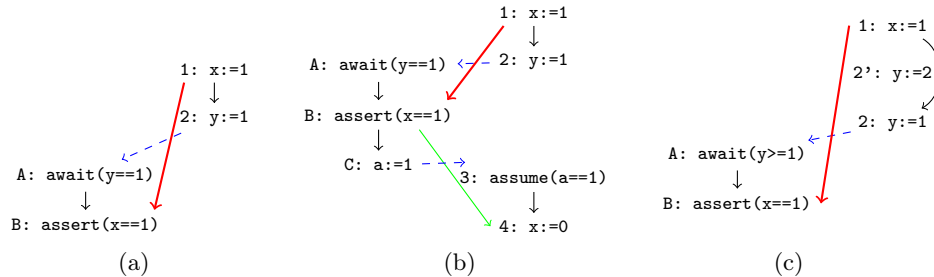

(a) (b) (c)

Fig. 3: Sample Good Traces for Regression-preventing constraints

be activated by both 2 and 2'; in the trace we analyse it is activated by 2. Moving 2' before 1 could activate the await "too early" and the assertion would fail (trace: $2' \to A \to B$). However, it is not possible to learn this purely with data-flow analysis – for example, if statement 2' was `y := -1`, then this would not lead to a bad trace. Hence, we exclude such cases from our definition of regressions by requiring that the await reads `A` reads from the same location.

Learning Sound Regression-Preventing Constraints. The sound regression-preventing constraint learned by our algorithm for a trace ensures that the data-flow into an assertion is preserved. This is achieved through two steps: suppose an assertion at location l_a reads from a write at location l_w. First, the constraint ensures that l_w always happens before l_a. Second, the constraint ensures that no other writes interfere with the above read-write relationship.

For ensuring happens-before relationships, we use the notion of a *cover*. Intuitively, given a trace π of P where location l_x happens before location l_y, we learn a Φ that ensures that if $P' \models \Phi$, then each trace π' of P' obtained as free and program transformations acting on π satisfies the happens-before relationship between l_x and l_y. Formally, given a trace π of program P, we call a path $x_1 \to x_2 \to \ldots \to x_n$ in the trace analysis graph a *cover* of edge $x \to y$ if $x = x_1 \wedge y = x_n$ and each of $x_i \to x_{i+1}$ is either a intra-thread order edge, or a data-flow into conditionals edge, or a non-free order edge.

Given a trace $\pi = S_0 l_0 S_1 l_1 \ldots S_n$, where statement at position r (i.e., l_r) reads a set of variables (say $\mathcal{V}$) written by a statement at position w (i.e., l_w), the the non-interference edges define a sufficient set of happens-before relations to ensure that no other statements can interfere with the read-write pair, i.e., that every other write to $\mathcal{V}$ either happens before w or after r. Formally, we have that $interfere(w \to r) = \{r \to w' \mid w' > r \wedge write(l_{w'}) \cap write(l_w) \cap Read(l_r) \neq \emptyset\} \cup \{w' \to w \mid w' < w \wedge write(l_{w'}) \cap write(l_w) \cap Read(l_r) \neq \emptyset\}$ where $Read(l)$ and $write(l)$ are the variables read and written at location l. If $w = \bot$, we have $interfere(w \to r) = \{r \to w' \mid w' > r \wedge write(l_{w'}) \cap Read(l_r) \neq \emptyset\}$.

Algorithm 1 Algorithm *LearnGoodUnder*

Require: A good trace π
Ensure: Regression-preventing constraint Φ
1: $\Phi \leftarrow true; G \leftarrow G(\pi)$
2: **for all** $e \in \left(DFAsserts(G) \cup \bigcup_{f \in DFAsserts(G)} interfere(f) \right)$ **do**
3:     **if** e is not covered **then return** $\bigwedge \{l_x \leq l_y \mid x \to y$ is a intra-thread order edge$\}$
4:     $\Phi' \leftarrow$ `false`
5:     **for all** $x_1 \to x_2 \to \ldots \to x_n$ cover of e **do**
6:         $\Phi' \leftarrow \Phi' \vee \bigwedge \{l_{x_i} \leq l_{x_{i+1}} \mid x_i \to x_{i+1}$ is a intra-thread order edge and $x_i \neq \bot$
                l_{x_i} and $l_{x_{i+1}}$ are from the same execution of a basic block in π }
7:     $\Phi \leftarrow \Phi \wedge \Phi'$
8: **return** Φ

Algorithm 1 works by ensuring that for each data-flow into assertions edge e, the edge itself is covered and that the interference edges are covered. For each such cover, the set of intra-thread order edges needed for the covering are

conjuncted to obtain a constraint. We take the disjunction Φ' of the constraints produced by all covers of one edge and add it to a constraint Φ to be returned. If an edge cannot be covered, the algorithm falls back by returning a constraint that fixes all current intra-thread orders. The algorithm can be made to run in polynomial time in $|\pi|$ using standard dynamic programming techniques.

Theorem 1. *Given a trace π, Algorithm 1 returns a constraint Φ that is a sound regression-preventing constraint for π and runs in polynomial time in $|\pi|$.*

Proof (Outline). The fallback case (line 3) is trivially sound. Let us assume towards contradiction that there is a bad trace $\pi' = S_0' l_0' S_1' l_1' \ldots S_n'$ of $P' \models \Phi$, that is obtained by transformation of $\pi = S_0 l_0 S_1 l_1 \ldots S_n$. For each $0 \leq i < n$, let $p(i)$ be such that the instruction at position i in π is at position $p(i)$ in π' after the sequence of switching transformations taking π to π'.

If for every data-flow into assertion edge in $x \to y$ in $G(\pi)$, we have that $p(x) \to p(y)$ is a corresponding data-flow into assertion edge in $G(\pi')$, then it can be easily shown that π' is also good (each corresponding edge in π' reads the same values as in π). Now, suppose $x \to y$ is the first (with minimal x) such edge in π that does not hold in π'. We will show in two steps that $p(x)$ happens before $p(y)$ in π', and that $p(y)$ reads from $p(x)$ which will lead to a contradiction.

For the first step, we know that there exists a cover of $x \to y$ in π. For now, assume there is exactly one cover – the other case is similar. For each edge $a \to b$ in this cover, no switching transformation can switch the order of l_a and l_b:

- If $a \to b$ is a data-flow into conditionals edge, as π' has to preserve all *DFConds* edges (definition of regression), $p(a)$ happens before $p(b)$ in π'.
- If $a \to b$ is a non-free order edge, no switching transformation can reorder a and b as that would change variables values (by definition of non-free edges).
- If $a \to b$ is a intra-thread order edge, we have that $P' \models \Phi$ and $\Phi \implies a \leq b$, and hence, no switching transformation would change the order of a and b.

Hence, we have that all the happens before relations given by the cover are all preserved by π' and hence, $p(a)$ happens before $p(a)$ in π'. The fact that $p(y)$ reads from $p(x)$ follows from a similar argument with the *interfere*$(x \to y)$ edges showing that every interfering write either happens before $p(x)$ or after $p(y)$. $\square$

3.2 Eliminating Bad Traces

Given a bad trace π of P, a program constraint Φ is a *error eliminating constraint* if for all transformations $\theta_1, \ldots, \theta_k$ and P' such that $P \xrightarrow{\theta_1 \ldots \theta_k} P'$ and $P' \models \Phi$, each bad trace π' in $\theta_k^f \circ \ldots \circ \theta_1^f(\pi)$ is not a trace of P'. In [4], we presented an algorithm to fix bad traces using reordering and atomic sections. The main idea behind the algorithm is as follows. Given a bad trace π, we (a) first, generalize the trace into a partial order trace; and (b) then, compute a program constraint that violates some essential part of the ordering necessary for the bug.

More precisely, the procedure builds a trace elimination graph which contain edges corresponding to the orderings necessary for the bug to occur, as well as the edges corresponding program constraints. Fixes are found by finding cycles
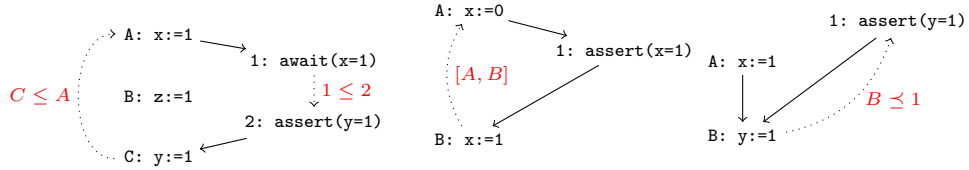
Fig. 4: Eliminating bad traces

in this graph – the conjunction of the program constraints in a cycle form an error elimination constraint. Intuitively, the program constraints in the cycle will enforce a happens-before conflicting with the orderings necessary for the bug.

Example 4. Consider the program in Figure 4(left) and the trace elimination graph for the trace $A; B; 1; 2; C$. The orderings A happens-before 1 and 2 happens-before C are necessary for the error to happen. The cycle $C \to A \to 1 \to 2 \to C$ is the elimination cycle. The corresponding error eliminating constraint is $C \leq A \wedge 1 \leq 2$, and one possible fix is to move C ahead of A. For the bad trace $A; 1; B$ in Figure 4(center), the elimination cycle is $A \to 1 \to B \to A$ giving us the constraint $[A; B]$ and an atomic section around $A; B$ as the fix.

The FixBad algorithm. The *FixBad* algorithm takes as input a program P, a constraint Φ and a bad trace π. It outputs a program constraint Φ', sequence of program transformations $\theta_1, \ldots, \theta_k$, and a new program P', such that $P \xrightarrow{\theta_1 \ldots \theta_k} P'$. The algorithm guarantees that (a) Φ' is an error eliminating constraint; (b) $P' \models \Phi \wedge P' \models \Phi'$; and (c) if there is no preemption-free trace π' of P such that π freely transforms to π' (i.e., $\pi' \in f(\pi)$), then none of the transformations $\theta \in \{\theta_1, \ldots, \theta_k\}$ acts across preemption-points. The fact that $\theta_1 \ldots \theta_k$ and P' can be chosen to satisfy (c) is a consequence of the algorithm described in [4].

Fixes using wait/notify statements. Some programs cannot be fixed by statement reordering or atomic section insertion. These programs are in general outside our definition of the program repair problem as they have bad sequential traces. However, they can be fixed by the insertion of wait/notify statements. One such example is depicted in Figure 4(right) where the trace $1; A; B$ causes an assertion failure. A possible fix is to add a `wait` statement before 1 and a corresponding `notify` statement after B. The algorithm *FixBad* can be modified to insert such wait-notify statements by also considering constraints of the form $X \preceq Y$ to represent that X is scheduled before Y – the corresponding program transfomation is to add a wait statement before Y and a notify statement after X. In Figure 4(right), the edge $B \to 1$ represents such a constraint $B \preceq 1$ – the elimination cycle $1 \to B \to 1$ corresponds to the above described fix.

4   The Program-Repair Algorithm

Algorithm 2 is a program-repair procedure to fix concurrency bugs while avoiding regressions. The algorithm maintains the current program P, and a constraint Φ that restricts possible reorderings. In each iteration, the algorithm tests if P is correct and if so returns P. If not it picks a trace π in P (line 4).

Algorithm 2 Program-Repair Algorithm for Concurrency

Require: A concurrent program P, all sequential traces are good
Ensure: Program P^* such that P^* has no bad traces
1: $\Phi \leftarrow true; T_G \leftarrow \emptyset$
2: **while true do**
3:    **if** $Verify(P) = $ **true then return** P
4:    Choose π from P (non-deterministic)
5:    **if** π is non-erroneous **then**
6:       $\Phi \leftarrow \Phi \wedge LearnGood(\pi); T_G \leftarrow T_G \cup \{\pi\}$
7:    **else**
8:       $([\theta_1, \ldots, \theta_k], P, \Phi') \leftarrow FixBad(P, \mathcal{P}, \Phi, \pi); \quad \Phi \leftarrow \Phi \wedge \Phi'$
9:       $T_G \leftarrow \bigcup_{\pi_g \in T_G} \{\pi_g' | \pi_g' \in \theta_k \circ \ldots \circ \theta_1(\pi^g) \wedge \pi_g' \in P\}$

If the trace is good it learns the regression-preventing constraint Φ for π and the trace π is added to the set of good traces T_G (T_G is required only for the correctness proof). If π is bad it calls *FixBad* to generate a new program that excludes π while respecting Φ, and Φ is strengthened by conjunction with the error elimination constraint Φ' produced by *FixBad*. The algorithm terminates with a valid solution for all choices of P' in line 8 as the constraint Φ is strengthened in each *FixBad* iteration. Eventually, the strongest program-constraint will restrict the possible program P' to one with large enough atomic sections such that it will have only preemption-free or sequential traces.

Theorem 2 (Soundness). *Given a program P, Algorithm 2 returns a program P' with no bad traces that preserves the sequential and correct preemption-free behaviour of P. Further, each iteration of the **while** loop where a bad trace π is chosen performs a regression-free error fix with respect to the good traces T_G.*

The extension of the *FixBad* algorithm to wait/notify fixes in Algorithm 2 may lead to P' not preserving the good preemption-free and sequential behaviours of P. However, in this case, the input P violates the pre-conditions of the algorithm.

Theorem 3 (Fair Termination). *Assuming that a bad trace will eventually be chosen in line 4 if one exists in P, Algorithm 2 terminates for any instantiation of FixBad.*

A Generic Program-Repair Algorithm. We now explain how our program-repair algorithm relates to generic synthesis procedures based on *counter-example guided inductive synthesis* (CEGIS) [17]. In the CEGIS approach, the input is a *partial-program* $\mathcal{P}$, i.e., a non-deterministic program and the goal is to specialize $\mathcal{P}$ to a program P so that all behaviours of P satisfy a specification. In our case, the partial-program would non-deterministically choose between various reorderings and atomics sections. Let $\mathcal{C}$ be the set of choices (e.g., statement orderings) available in $\mathcal{P}$. For a given $\mathbf{c} \in \mathcal{C}$, let $\mathbb{P}(\mathcal{P}, \mathbf{c}, \mathbf{i})$ be the predicate that program obtained by specializing $\mathcal{P}$ with $\mathbf{c}$ behaves correctly on the input $\mathbf{i}$.

The CEGIS algorithm maintains a set $\mathcal{E}$ of inputs called experiments. In each iteration, it finds $\mathbf{c}^* \in \mathcal{C}$ such that the $\forall \mathbf{i} \in \mathcal{E} : \mathbb{P}(\mathcal{P}, \mathbf{c}^*, \mathbf{i})$. Then, it attempts to find an input $\mathbf{i}^*$ such that $\mathbb{P}(\mathbf{c}^*, \mathbf{i}^*)$ does not hold. If there is no such input, then

$\mathbf{c}^*$ is the correct specialization. Otherwise, $\mathbf{i}^*$ is added to $\mathcal{E}$. This procedure is illustrated in Figure 5(left). Alternatively, CEGIS can be rewritten in terms of constraints on $\mathcal{C}$. For each input $\mathbf{i}$, we associate the constraint $\phi_\mathbf{i}$ where $\phi_\mathbf{i}(\mathbf{c}) \Leftrightarrow \mathbb{P}(\mathcal{P}, \mathbf{c}, \mathbf{i})$. Now, instead of $\mathcal{E}$, the algorithm maintains the constraint $\Phi = \bigwedge_{\mathbf{i} \in \mathcal{E}} \phi_\mathbf{i}$. Every iteration, the algorithm picks a $\mathbf{c}$ such that $\mathbf{c} \models \Phi$; tries to find an input $\mathbf{i}^*$ such that $\neg\mathbb{P}(\mathcal{P}, \mathbf{c}, \mathbf{i})$ holds, and then strengthens Φ by $\phi_{\mathbf{i}^*}$.
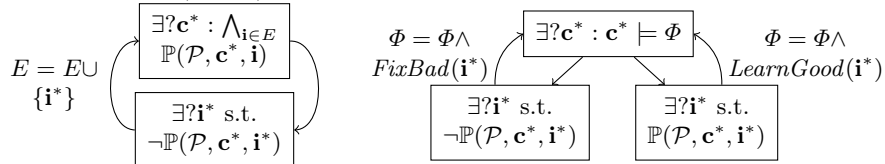

Fig. 5: The CEGIS and PACES spectrum

This procedure is exactly the else branch (i.e., *FixBad* procedure) of an iteration in Algorithm 2 where $\mathbf{i}^*$ and $\phi_{\mathbf{i}^*}$ correspond to π and $FixBad(\pi)$. Intuitively, the initial variable values in π and the scheduler choices are the inputs to our concurrent programs. This suggests that the then branch in Algorithm 2 could also be incorporated into the standard CEGIS approach. This extension (dubbed PACES for *Positive and Counter-Examples in Synthesis*) to the CEGIS approach is shown in Figure 5(right). Here, the algorithm in each iteration may choose to find an input for which the program is correct and use the constraints arising from it. We discuss the advantages and disadvantages of this approach below.

Constraints vs. Inputs. A major advantage of using constraints instead of sample inputs is the possibility of using over- and under-approximations. As seen in Section 3.1, it is sometimes easier to work with approximations of constraints due to simplicity of representation at the cost of potentially missing good solutions. Another advantage is that the sample inputs may have no simple representations in some domains. The scheduler decisions are one such example – the scheduler choices for one program are hard to translate into the scheduler choices for another. For example, the original CEGIS for concurrency work [16] uses ad-hoc trace projection to translate the scheduler choices between programs.

Positive-examples and Counter-examples vs. Counter-examples. In standard program-repair tasks, although the faulty program and the search space $\mathcal{C}$ may be large, the solution program is usually "near" the original program, i.e., the fix is small. Further, we do not want to change the given program unnecessarily. In this case, the use of positive examples and over-approximations of learned constraints can be used to narrow down the search space quickly. Another possible advantage comes in the case where the search space for synthesis is structured (for example, in modular synthesis). In this case, we can use the correct behaviour displayed by a candidate solution to fix parts of the search space.

5 Implementation and Experiments

We implemented Algorithm 2 in our tool ConRepair. The tool consists of 3300 lines of Scala code and is available at https://github.com/thorstent/ConRepair.

Model checker CBMC [1] is used for generating both good and bad traces, and on an average more than 95% of the total execution time is spent in CBMC. Model checking is far from optimal to obtain good traces, and we expect that techniques from [13] can be used to generate good traces much faster. Our tool can operate in two modes: In "mixed" mode it first analyses good traces and then proceeds to fixing the program. The baseline "badOnly" mode skips the analysis of good traces (corresponds to the algorithm in [4]).

In practice the analysis of bad traces usually generates a large number of potential reorderings that could fix the bug. Our original algorithm from [4] (badOnly `ce1`) prefers reorderings over atomic sections, but in examples where an atomic section is the only fix, this algorithm has poor performance. To address this we implemented a heuristic (`ce2`) that places atomic sections before having tried all possible reorderings, but this can result in solutions having unnecessary atomic sections.

The fall back case in Algorithm 1 severely limits further fixes – it forces further fixes involving the same instructions to be atomic sections. Hence, in our implementation, we omit this step and prefer an unsound algorithm (i.e., not necessarily regression-free) that can fix more programs with reorderings. While the implemented algorithm is unsound, our experiments show that even without the fallback, in our examples, there is no regression except for one artificial example (`ex-regr.c`) constructed precisely for that purpose.

Benchmarks. We evaluate our tool on a set of examples that model real bugs found and fixed in Linux device drivers by their developers. To this end, we explored a history of bug fixes in the drivers subtree of the Linux kernel and identified concurrency bugs. We further focused our attention on a subset of particularly subtle bugs involving more than two racing threads and/or a mix of different synchronization mechanisms, e.g., lock-based and lock-free synchronization. Approximately 20% of concurrency bugs that we considered satisfy this criterion. Such bugs are particularly tricky to fix either manually or automatically, as new races or deadlocks can be easily introduced while eliminating them. Hence, these bugs are most likely to benefit from good trace analysis.

Table 1 shows our experimental results: the iterations and the wall-clock time needed to find a valid fix for our mixed algorithm and the two heuristics of the badOnly algorithm. For the mixed algorithm the time is split into the time needed to generate and analyse good traces (first number) and the time needed for the fixing afterwards.

File	LOC	mixed	badOnly ce1	badOnly ce2
`ex1.c`	60	1	2	2
`ex2.c`	37	2	5	6
`ex3.c`	35	1	2	2
`ex4.c`	60	1	2	2
`ex5.c`	43	1	8	3
`ex-regr.c`	30	2	2	2
`paper1.c`	28	1	3	3^a
`dv1394.c`	81	1 (13+4s)	51 (60s)	5^a (9s)
`iwl3945.c`	66	1(3+2s)	2(2s)	2(2s)
`lc-rc.c`	40	10 (2+7s)	179 (122s)	203 (134s)
`rtl8169.c`	405	7 (10+45m)	>100 (>6h)	8 (54m)
`usb-serial.c`	410	4 (56+20m)	6 (38m)	6 (38m)

Table 1: Results in iterations and time needed.

^{a}`ce2` heuristic placed unnecessary atomic section

Detailed analysis. The ar-
tificial examples `ex1.c` to `ex5.c` are used for testing and take only a few seconds;
example `paper1.c` is the one in Figure 1a. Example `ex-regr.c` was constructed
to show unsoundness of the implementation. Example `usb-serial.c` models the
USB-to-serial adapter driver. Here, from the good traces the tool learns that two
statements should not be reordered as it will trigger another bug. This prompts
them to be reordered above a third statement together, while the badOnly anal-
ysis would first move one, find a new bug, and then fix that by moving the other
statement. Thus, the good trace analysis saves us two rounds of bug fixing and
reduces bug fixing time by 18 minutes.

The `rtl8169.c` example models the Realtek 8169 driver containing 5 concur-
rency bugs. One of the reorderings that the tool considers introduces a new bug;
further, after doing the reordering, the atomic section is the only valid fix. The
good trace analysis discover that the reordering would lead to a new bug, and
thus does the algorithm does not use it. But, without good traces, the tool uses
the faulty reordering and then `ce1` takes a very long time to search through all
possible reorderings and then discover that an atomic section is required. The
situation is improved when using heuristic `ce2` as it interrupts the search early.
However, the same heuristic has an adverse effect in the `dv1394.c` example: by
interrupting the search early, it prevents the algorithm from finding a correct re-
ordering and inserts an unnecessary atomic section. The `dv1394.c` example also
benefits from good traces in a different way than the other examples. Instead
of preventing regressions, they are used to obtain *hints* as to what reorderings
would provide coverage for a specific data-flow into assertion edge. Then, if a
bad trace is encountered and can be fixed by the hinted reordering, the hinted
reordering is preferred over all other possible ones. Without hints the `dv1394.c`
example would require 5 iterations. Though hints are not part of our theory they
are a simple and logical extension.

Example `lc-rc.c` models a bug in an ultra-wide band driver that requires
two reorderings to fix. Though there is initially no deadlock, one may easily be
introduced when reordering statements. Here, the good-trace analysis identifies
a dependency between two `await` statements and learns not to reorder state-
ments to prevent a deadlock. Without good traces, a large number of candidate
solutions that cause a regression are generated.

6   Conclusion

We have developed a regression-free algorithm for fixing errors that are due to
concurrent execution of the program. The contributions include the problem
setup (the definitions of program repair for concurrency, and the regression-free
algorithm), the PACES approach that extends the CEGIS loop with learning
from positive examples, and the analysis of positive examples using data flow to
assertions and to synchronization constructs.

There are several possible directions for future work. One interesting direc-
tion is to examine the possibility of extending the definition of regressions (see
Remark 1 and Example 3) – this requires going beyond data-flow analysis for

learning regression-preventing constraints. Another possible extension is to remove the assumption that the errors are data-independent. A more pragmatic goal would be to develop a practical version of the tool for device-driver synthesis starting from the current prototype.

Acknowledgements. We would like to thank Daniel Kroening and Michael Tautschnig for their prompt help with all our questions about CBMC. We would also like to thank Roderick Bloem, Bettina Könighofer and Roopsha Samanta for fruitful discussions regarding repair of concurrent programs.

References

1. CBMC, http://www.cprover.org/cbmc/
2. Alglave, J., Kroening, D., Tautschnig, M.: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In: CAV '13
3. Alur, R., Bodík, R., Juniwal, G., Martin, M., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. pp. 1–17 (2013)
4. Černý, P., Henzinger, T., Radhakrishna, A., Ryzhyk, L., Tarrach, T.: Efficient synthesis for concurrency by semantics-preserving transformations. In: CAV. pp. 951–967 (2013)
5. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI '08
6. Deshmukh, J., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Logical Concurrency Control from Sequential Proofs. In: LMCS (2010)
7. von Essen, C., Jobstmann, B.: Program repair without regret. In: CAV. pp. 896–911 (2013)
8. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: POPL. pp. 129–142 (2013)
9. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to C. In: CAV. pp. 358–371 (2006)
10. Jin, G., Zhang, W., Deng, D., Liblit, B., Lu, S.: Automated Concurrency-Bug Fixing. In: OSDI '12
11. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: CAV. pp. 226–238 (2005)
12. Samanta, R., Deshmukh, J., Emerson, A.: Automatic generation of local repairs for boolean programs. In: FMCAD. pp. 1–10 (2008)
13. Sen, K.: Race Directed Random Testing of Concurrent Programs. In: PLDI '08
14. Sinha, N., Wang, C.: On Interference Abstractions. In: POPL '11
15. Sinha, N., Wang, C.: Staged concurrent program analysis. In: FSE '10
16. Solar-Lezama, A., Jones, C., Bodík, R.: Sketching concurrent data structures. In: PLDI. pp. 136–148 (2008)
17. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS '06
18. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL '10
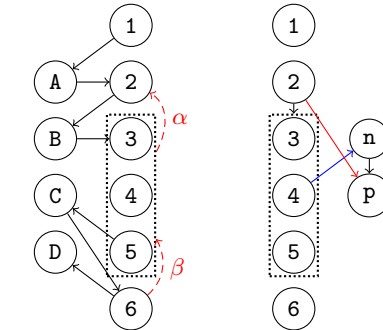
```
                    iwl3945_bg_alive_start_thread()
                    {
config_thread () {
  A: lock(rtnl_lock);        1: lock(lock);          reassoc_thread () {
  B: lock(lock);             2: restart = 1;           n: await(notify==1);
  C: unlock(lock);         ⎡3: lock(rtnl_lock);⎤       p: assert(restart==1);
  D: unlock(rtnl_lock);    ⎢4: notify = 1;     ⎥     }
}                          ⎣5: unlock(rtnl_lock);⎦
                             6: unlock(lock);
                    }
```

(a) Simplified threads of the iwl3945 driver

(b) Possible fixes for this trace

(c) Learning from a good trace

Fig. 6: Model of deadlock bug in the iwl3945 driver

A   The iwl3945 driver.

This example models a real concurrency bug found in the Linux driver for the Intel wireless adapter 3945. The model involves three kernel threads calling driver entry points shown in Figure 6a.

The driver suffers from a classical ABBA deadlock; if the `config_thread` locks `rtnl_lock` then the `iwl3945_bg_alive_start_thread` locks `lock` none of the threads can proceed.

`config_thread` is located outside the driver and cannot be changed in order to fix the bug. Furthermore instructions `3` to `5` need to stay together because they are located in a separate function in the original code. The `iwl3945_bg_alive_start_thread` is used to restart the device if it no longer responds properly. After a restart (line `2`) the `reassoc_thread` is notified (line `4`), which depends on the device having completed the restart.

In the actual model the deadlock is modelled with an assertion between lines `A` and `B` that tests which thread owns which locks and which thread is waiting for which locks. The assertion fails if there is a deadlock. This construction is needed because CBMC does not support deadlock detection.

Our old algorithm `ce1` without a learning from good traces phase now proceeds to finding a bad trace, such as $1 \rightarrow A \rightarrow 2 \rightarrow B \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow C \rightarrow$

$6 \to D$ displayed in Fig. 6b. Note that after an assertion failure the trace is completed as if the assertion had succeeded in order to find all reordering options. This leaves a number of bug-fixes to the algorithm: We denote only the two important for the description of our algorithms. The first is to move the block 3-5 in front of 1 (denoted as α) and the second is to move 6 in front of 3-5 (denoted as β). Without further knowledge the algorithm will choose α for no particular reason. This does not fix the deadlock, but it furthermore introduces a regression because now assertion p may fail because the notify signal is sent before the restart is completed.

The good trace analysis prevents a regression with respect to good traces such as the trace where all threads are run sequentially. The relevant parts of the result of our analysis are depicted in Fig. 6c. The red and blue edge indicate the variable assignments the assertion and the await read from respectively. In order to protect the red edge two black edges are needed, from 2 to the instruction block 3-5 and from n to p. When the algorithm comes to the phase of fixing bugs it will discover the same two possibilities, but α is blocked by the black edge from 2 to 3-5. The algorithm will then choose the correct fix β.

This is also the fix the developers took in the actual driver. By taking the notify with its needed rtnl_lock out of the lock environment the deadlock is avoided.