Efficient Synthesis for Concurrency by Semantics-Preserving Transformations*

Pavol Černý[1], Thomas A. Henzinger[2], Arjun Radhakrishna[2], Leonid Ryzhyk[3], and Thorsten Tarrach[2]

[1] University of Colorado Boulder
[2] IST Austria
[3] NICTA

Abstract. We develop program synthesis techniques that can help programmers fix concurrency-related bugs. We make two new contributions to synthesis for concurrency, the first improving the efficiency of the synthesized code, and the second improving the efficiency of the synthesis procedure itself. The first contribution is to have the synthesis procedure explore a variety of (sequential) *semantics-preserving program transformations*. Classically, only one such transformation has been considered, namely, the insertion of synchronization primitives (such as locks). Based on common manual bug-fixing techniques used by Linux device-driver developers, we explore additional, more efficient transformations, such as the reordering of independent instructions. The second contribution is to speed up the counterexample-guided removal of concurrency bugs within the synthesis procedure by considering *partial-order traces* (instead of linear traces) as counterexamples. A partial-order error trace represents a set of linear (interleaved) traces of a concurrent program all of which lead to the same error. By eliminating a partial-order error trace, we eliminate in a single iteration of the synthesis procedure all linearizations of the partial-order trace. We evaluated our techniques on several simplified examples of real concurrency bugs that occurred in Linux device drivers.

1 Introduction

We develop program synthesis techniques that can help programmers fix concurrency-related bugs. We place ourselves into a setting all the threads of the program are sequentially correct, i.e., all the errors are due to concurrency. In this setting, our goal is to automatically fix concurrency errors. In other words, the programmer needs to worry only about sequential correctness, and the synthesis tool automatically makes the program safe for concurrent execution.

Our first contribution is to have the synthesis procedure explore a variety of (sequential) *semantics-preserving program transformations* to obtain efficient concurrent code. In existing work on partial program synthesis, mostly only one

* This work was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering), by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling), and by a gift from Intel Corporation.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 951–967, 2013.

such transformation has been considered: the insertion of synchronization primitives such as locks [15,3]. Our study of real-world concurrency bugs from device drivers shows that only 17% of bugs are fixed using locks. For the remaining bugs, developers use other program transformations that avoid the use of synchronization primitives yielding more efficient code. In particular, the most common fix used for 28% of driver concurrency bugs is *instruction reordering*, i.e., a rearrangement of program instructions that changes the driver's concurrent behavior while preserving the sequential semantics of each thread. For example, a pointer initialization may be moved before an instruction releasing another thread that dereferences the pointer. We develop a technique for automating this type of transformation. We also consider other semantics-preserving transformations inspired by practical bug-fixing techniques. For example, the synthesis tool may repeat idempotent instructions multiple times (we give an example where duplicating an instruction removes a concurrency bug).

Our second contribution is to increase the efficiency of the synthesis procedure itself by considering *partial-order traces* (as opposed to linear traces) as counterexamples in the context of counterexample-guided synthesis. A partial order on the instructions involved in the counterexample represents a set of linear counterexample traces that all lead to the same error. We first find a linear counterexample trace using an off-the-shelf tool, and then generalize it to a partial-order trace. We achieve this generalization by combining ideas from Lipton reduction [9] and error invariants [6]. We relax the ordering of a pair of instructions in the linear trace if swapping these instructions preserves error invariants (and thus the bug can still be reached). Intuitively, the resulting partial-order trace captures the 'true cause' of the bug. For instance, if the linear counterexample includes context switches that are not necessary to reach the bug, these context switches will not be required by the partial-order trace.

A key insight in our algorithm is that given the partial-order trace μ, the problem of eliminating μ can be phrased as the problem of creating a minimal cycle in a graph (representing the partial order) by adding new edges. A graph with a cycle does not allow linearization and hence a cycle corresponds to a set of transformations that together eliminate μ. The additional edges correspond to possible instruction reordering or the insertion of atomic sections. Each additional edge is labeled by a cost (for instance, the length of the atomic section).

We implemented our techniques in a prototype tool called ConcurrencySwapper. As specifications, we handle assertions, deadlocks, and generic conditions such as pointer use before initialization. However, our techniques apply to a larger class of reachability properties. For finding buggy traces, we use the model checker Poirot [1]. If Poirot produces a buggy trace, ConcurrencySwapper generalizes it to a partial-order trace, which it then tries to eliminate first by instruction reordering, and failing that, using an atomic section. Otherwise, the current version of the driver is returned, with all the discovered bugs fixed.

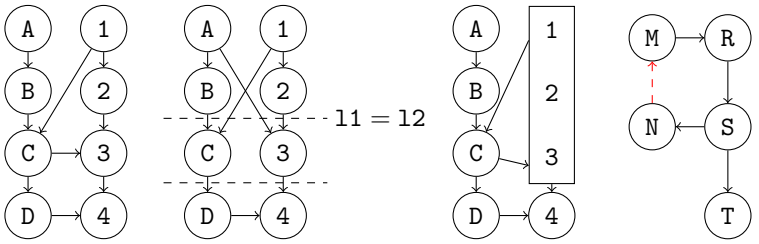
We evaluated our tool on (a) five microbenchmarks that are simplified versions of bugs from Linux device drivers, and (b) a simplified driver for the Realtek 8169 Ethernet controller. The latter had 364 LOC, seven threads, and contained

```
init: x = 0; t1 = F
thread1      thread2
A: l1 = x    1: l2 = x
B: l1++      2: l2++
C: x = l1    3: x = l2
D: t1 = T    4: assert(!t1∨x=2)
```

(a) Concurrent increment

```
init: IntrMask = 0; ready = 0; handled = 0;
init_thread        intr_thread
M: IntrMask = 1    R: assume(IntrMask = 1)
N: ready = 1       S: handled = ready
                   T: assert(handled)
```

(b) Interrupt handling



(c) $\preceq_\pi^1$ (d) $\preceq_\pi^2$ (e) $\preceq_\pi^{1+}$ (f) $\preceq_\theta$

Fig. 1. Illustrative examples

five bugs. In the experiments, we found that: (a) bug finding and verification (in Poirot) dominates time spent generalizing counterexamples, and (b) using generalized counterexamples reduces the number of bug-finding iterations.

Related Work. Synthesis for concurrent programs has attracted considerable research [13,15,3], which is mainly concerned with the insertion of locks. In contrast, we consider general semantics-preserving transformations, a key one being instruction reordering. In [13] and [14], an order for a given set of instructions is synthesized, whereas we are given a buggy program, and we reorder instructions to remove the bug (while preserving the sequential semantics). The main difference from previous work is the algorithm. We generalize counterexample traces to partial-order traces and eliminate them by adding constraints on the instruction order. In contrast, e.g., in [13] the problem of choosing orderings is reduced to resolution of nondeterminism.

Concurrent trace theory has long studied the idea of treating traces as partial orders over events, as in the seminal work on Mazurkiewicz traces (see, for example, [10]). However, their use in counterexample-guided approaches to abstraction refinement, verification, or synthesis has not been considered before. We augment concurrent traces with error invariants, on which there has been recent work in the sequential setting [6].

2 Illustrating Examples

Generalizing Buggy Traces. In Figure 1a, `thread1` and `thread2` concurrently increment `x`. The assertion states that `x` is 2 in the end. It fails in trace $\pi \equiv \mathtt{A} \to \mathtt{B} \to 1 \to \mathtt{C} \to 2 \to \mathtt{D} \to 3 \to 4$, where both threads read the initial `x` value 0, and then write back 1 to `x`. However, π is just one trace exhibiting this bug. For example, swapping `B` and `1` in π gives another buggy trace. Let $\preceq_\pi$ be a

total order where $\texttt{X} \preceq_\pi \texttt{Y}$ iff statement X occurs before Y in π. We relax $\preceq_\pi$ by removing all constraints $\texttt{X} \preceq_\pi \texttt{Y}$ where `X; Y` has the same effect as `Y; X`. This gives us the partial order $\preceq^1_\pi$ (shown in Figure 1c). All traces where the execution order respects $\preceq^1_\pi$ fail the assertion.

For `C` and `3`, the sequence `C; 3` is not equivalent to `3; C` when $\texttt{l1} \neq \texttt{l2}$. However, in all traces of $\preceq^1_\pi$, it can be seen that $\texttt{l1} = \texttt{l2} = 1$, and further, this is sufficient to trigger the bug. These sufficient conditions to trigger bugs are *error invariants*. Using this information, we can further relax $\preceq_\pi$ to $\preceq^1_\pi$ shown in Figure 1d, where the only constraints are that both threads read `x` before either writes to it, and that `D` occurs before `4`. A main component of our synthesis algorithm is the generalization of buggy traces to determine their root cause.

Atomic Sections. We attempt to eliminate the bug represented by $\preceq_\pi$ by adding atomic sections. For example, adding an atomic section around `1`, `2`, and `3` in $\preceq^1_\pi$ gives us $\preceq^{1+}_\pi$ from Figure 1e, where the atomic section is collapsed into a single node. Note that $\preceq^{1+}_\pi$ is not a valid partial order, as there is a cycle of nodes `[1;2;3]` and `C`. Intuitively, the cycle implies that `[1;2;3]` happens both before and after `C`, which is impossible. Hence, adding an atomic section around `[1;2;3]` eliminates all traces represented by $\preceq^1_\pi$ from the program. The atomic section `[1;2;3]` does not eliminate the buggy trace $\texttt{A} \to [1;2;3] \to \texttt{B} \to \texttt{C} \to \texttt{D} \to \texttt{4}$. Analyzing this trace similarly, we find that another atomic section `[A;B;C]` is needed to obtain a correct program.

The number of bug fixing iterations can be reduced using error invariants. For example, in $\preceq^2_\pi$, the atomic section `[1;2;3]` is not sufficient to create a cycle; instead, we immediately see that both `[1;2;3]` and `[A;B;C]` are needed.

Instruction Reordering. The example in Figure 1b is inspired by a real bug from a Linux device driver. Thread `intr_thread` runs when interrupts are enabled, i.e., `IntrMask` is 1, and attempts to handle them; it fails if the driver is not ready. The `init_thread` enables interrupts and readies the driver.

The bug is that interrupts are enabled before the driver is ready, for example, in trace $\theta \equiv \texttt{M} \to \texttt{R} \to \texttt{S} \to \texttt{N} \to \texttt{T}$. Note that statements `M` and `N` are independent, i.e., `M; N` is equivalent to `N; M`. We construct a partial order from θ as before, but remove the constraint $\texttt{M} \preceq_\theta \texttt{N}$, giving us Figure 1f (excluding the dashed edge). Adding the edge $\texttt{N} \to \texttt{M}$ creates a cycle and eliminates the bug. This edge changes the order of `M` and `N`, forcing the order `N; M`. This results in a correct program with the driver ready to handle interrupts before they are enabled.

Following the ideas presented in this section, our synthesis algorithm works by generalizing linear counterexample traces to partial-order traces and eliminating them using atomic section insertion or instruction reordering.

3 Model and Problem Statement

Let V be a set of variables ranging over a domain $\mathcal{D}$. A *V-valuation* is a function $\mathcal{V} : V \to \mathcal{D}$. A *state assertion* ϕ is a first-order constraint over valuations of variables. We model an *instruction* as an assertion τ over V and V'. Intuitively,

V and V' represent the values of the variables before and after the execution of the instruction, respectively. For example, $x' = x + y$ represents `x = x + y` in a C-like language. Given a V-valuations $\mathcal{V}$ and a V'-valuation $\mathcal{V}'$, we write $(\mathcal{V}, \mathcal{V}') \models \tau$ to denote the fact that assertion τ holds for values given by $\mathcal{V}$ and $\mathcal{V}'$. Furthermore, we require that the instruction is deterministic, i.e., for every $\mathcal{V}$, there is at most one $\mathcal{V}'$ such that $(\mathcal{V}, \mathcal{V}') \models \tau$.

We model procedures as *control-flow graphs* (CFGs) with locations labeled with instructions. Formally, a *method* is a tuple $\langle V, I, O, S, \Delta, s_\iota, s_f, inst\rangle$ where (a) V is a set of variables and $I \subseteq V$ and $O \subseteq V$ are *input* and *output* variables respectively; (b) S is a finite set of control locations and s_ι and s_f are the initial and final control locations respectively; (c) $\Delta \subseteq S \times S$ is a set of transitions; and (d) *inst* is a function labeling control locations with instructions. The initial values of the input variables and the final values of the output variables are the arguments and the return values of the method call, respectively. We assume that methods are deterministic, and that all the CFGs are reducible (see [11]).

A *concurrent library* $\mathcal{P}$ is a tuple $\langle M_1, \ldots, M_n\rangle$ of methods with mutually disjoint control locations. Let $locs(M_i)$, $vars(M_i)$, $\Delta(M_i)$ be the control locations, variables, and transitions of M_i, respectively. Further, let $globals(\mathcal{P})$ denote the variables shared among methods, and $locs(\mathcal{P}) = \bigcup_i locs(M_i)$ the locations of $\mathcal{P}$.

Modeling language constructs. Programs are encoded as CFGs in a standard way. For example, `if(x == 0)` is a choice between then and else branches prefixed with `assume(x == 0)` and `assume(x != 0)` respectively. We model assertions in M_i using variable err_{M_i} that is set to 1 when an assertion fails. To block execution after an assertion failure, we replace every instruction τ by $err_{M_i} = 0 \wedge \tau$. Atomic sections are modeled using an auxiliary variable that prevents other methods from running when the program is inside an atomic section.

Semantics. The methods of a library can be executed in parallel, by an unbounded number of threads. We assume that each thread executes one method. Let *Tids* be a set of thread identifiers. A *thread state* is a triple $(tid, s, \mathcal{V})$ where $tid \in Tids$, $s \in locs(M_i)$ is a control location and $\mathcal{C}$ is a $(vars(M_i) \setminus globals(\mathcal{P}))$-valuation. A thread state is initial (resp., final) if the control state is initial (resp. final). A *library state* $(\mathcal{G}, \mathcal{T})$ contains a $globals(\mathcal{P})$-valuation $\mathcal{G}$, and a set $\mathcal{T}$ of thread states with unique thread identifiers. State $(\mathcal{G}, \mathcal{T})$ is final for thread *tid* if $(tid, s, \mathcal{V}) \in \mathcal{T}$ where s is a final control location. We denote by $(\mathcal{G}, \mathcal{T})_{\restriction tid}$ the valuation given by $\mathcal{G} \cup \mathcal{V}$ where $\mathcal{V}$ is such that $(tid, s, \mathcal{V}) \in \mathcal{T}$.

A *single-step execution* of thread *tid* is a triple $((\mathcal{G}, \mathcal{T}), s, (\mathcal{G}', \mathcal{T}'))$ such that there exist M_i, $(tid, s, \mathcal{V}) \in \mathcal{T}$ and $(tid, s', \mathcal{V}') \in \mathcal{T}'$ with (a) $\mathcal{T} \setminus \{(tid, s, \mathcal{V})\} = \mathcal{T}' \setminus \{(tid, s', \mathcal{V}')\}$; and (b) $(\mathcal{G} \cup \mathcal{V}, \mathcal{G}' \cup \mathcal{V}') \models inst(s)$ and $(s, s') \in \Delta(M_i)$. A *trace* π of $\mathcal{L}$ is a sequence $(\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1)s_1 \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ where every thread state in $\mathcal{T}_0$ is initial, and every $((\mathcal{G}_i, \mathcal{T}_i), s_i, (\mathcal{G}_{i+1}, \mathcal{T}_{i+1}))$ is a single-step execution. Since our instructions are deterministic, we write π as $[(\mathcal{G}_0, \mathcal{T}_0)]s_0 \to s_1 \to \ldots$.

A *sequential trace* of $\mathcal{L}$ is a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0 \ldots$ such that all the transitions of a single thread *tid* occur in a contiguous block (say $(\mathcal{G}_k, \mathcal{T}_k)s_k \ldots (\mathcal{G}_{k+l}, \mathcal{T}_{k+l})$) with $(\mathcal{G}_{k+l}, \mathcal{T}_{k+l})$ being final for *tid*. For any such block, we write

$[(\mathcal{G}_k, \mathcal{I}_k) - seq(M_i) \rightarrow (\mathcal{G}_{k+l}, \mathcal{O}_{k+l})]$ where $\mathcal{I}_k$ and $\mathcal{O}_{k+l}$ are the valuation of the input and output variables in $(\mathcal{G}_k, \mathcal{S}_k)_{\uparrow tid}$ and $(\mathcal{G}_{k+l}, \mathcal{S}_{k+l})_{\uparrow tid}$, respectively. The *sequential semantics* $SeqSem(M_i)$ of method M_i is the set of all relations $[(\mathcal{G}, \mathcal{I}) - seq(M_i) \rightarrow (\mathcal{G}', \mathcal{O})]$ that occur in sequential traces of $\mathcal{P}$. Intuitively, the $SeqSem(M_i)$ characterizes all sequential input-output behaviour of a method.

For every trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0 \dots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$, we define $time_\pi : \{0, \dots, n\} \rightarrow \mathbb{N}$ as follows: (a) $time_\pi(0) = 0$; (b) $time_\pi(i+1) = time_\pi(i)$ if s_i is in an atomic section; and (c) $time_\pi(i+1) = time(i) + 1$ otherwise.

The Synthesis Problem. A trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \dots$ of $\mathcal{P}$ is *erroneous* if some library state $(\mathcal{G}_i, \mathcal{T}_i)$ has an err_{M_i} set to 1. Let $\mathcal{P} = \langle M_1, \dots, M_n \rangle$ be a concurrent library. Library $\mathcal{P}$ is *sequentially correct* if every sequential trace of $\mathcal{P}$ is not erroneous. Let M and M' be two methods with the same input and output variables. M and M' are sequentially equivalent, if $SeqSem(M) = SeqSem(M')$.

Let $\mathcal{P}' = \langle M'_1, \dots, M'_n \rangle$ be a concurrent library. We say that $\mathcal{P}'$ is *sequentially equivalent* to $\mathcal{P}$ if, for all i, M_i is sequentially equivalent to M'_i.

The concurrent library synthesis problem asks the following: given a sequentially correct concurrent library $\mathcal{P} = \langle M_1, \dots, M_n \rangle$, output a sequentially equivalent library $\mathcal{P}' = \langle M'_1, \dots, M'_n \rangle$ such that every trace of $\mathcal{P}'$ is not erroneous. Intuitively, the problem asks for a version $\mathcal{P}'$ of $\mathcal{P}$ which is safe for concurrent execution, but has the same sequential semantics. The obvious solution involving adding an atomic section around each method is undesirable. Instead, our approach is as follows: (a) find an erroneous trace π of $\mathcal{P}$; (b) compute a generalization of π; and (c) transform the methods minimally to avoid the bug.

We observe the following about the definition. First, we assume a sequentially correct library to ensure that at least one solution to the synthesis problem exists. However, our approach can be easily extended to not have this assumption; in that case, it may terminate without finding a solution. Second, correctness was specified by assertions in the code. However, our approach works for a more general class of specifications, namely, for safety properties on the global state.

4 Semantics-Preserving Transformations

We present a few sequential-semantics preserving transformations, focusing on statement reordering and insertion of atomic sections. We also give a motivating example for idempotent statement duplication, but do not treat it formally, in the interest of space. Our synthesis algorithm operates by collecting constraints on possible solutions, and hence, we present representations of constraints on possible solutions obtained by the considered transformations.

Statement Reordering. Statement reordering is a transformation that changes the order of statements within a method. Notably, this transformation can change concurrent behavior without changing sequential semantics (e.g., in Figure 1b). A *block* is a single-entry, maximal sequence of control locations $s_0 \dots s_k$ representing straight-line code. For simplicity, we only consider reorderings within blocks. Our techniques can be extended to allow reorderings across block boundaries.

We represent multiple reorderings of method M compactly using a *reordering constraint* $\sqsubseteq \subseteq locs(M) \times locs(M)$, that specifies a partial order on $locs(M)$. We may have $s \sqsubseteq s'$ only if s and s' are from the same block. We write $s \sqsubsetneq s'$, if $s \sqsubseteq s'$ and s is different from s'. Let M' be a method obtained by statement reordering from method M. Method M'satisfies $\sqsubseteq$ if for all $s \sqsubsetneq s'$, s occurs before s' in the corresponding block. A reordering constraint $\sqsubseteq$ is *weaker* than $\sqsubseteq'$ if $s \sqsubseteq s' \implies s \sqsubseteq' s'$.

As our reorderings need to preserve sequential semantics, we can compute some reordering constraints even before considering concurrent executions. The procedure `SemPreservingOrders` computes a reordering constraint $\sqsubseteq$ as follows. It first constructs the total order $\sqsubseteq$ of control states in each block. Then, it picks s and s' such that $s \sqsubsetneq s'$, and checks if $inst(s)$ and $inst(s')$ commute, i.e., we test using a theorem prover two conditions: (a) $inst(s'); inst(s)$ can execute to completion from each state $inst(s); inst(s')$ can; and (b) they have the same effect. If they commute, then the pair (s, s') is removed from $\sqsubseteq$ and we repeat the process. When not such pair exists, we return $\sqsubseteq$. If `SemPreservingOrders` returns $\sqsubseteq$ on input M, then every M' satisfying $\sqsubseteq$ (and obtained by reordering) is sequentially equivalent to M, and no weaker $\sqsubseteq'$ has the same property.

Example 1. Running `SemPreservingOrders` on the code fragment from Figure 1b gives us a single constraint $\texttt{S} \sqsubseteq \texttt{T}$ as all other pairs of statements are independent of each other. In Figure 1a, we get $\texttt{A} \sqsubseteq \texttt{B} \sqsubseteq \texttt{C}$ and $\texttt{1} \sqsubseteq \texttt{2} \sqsubseteq \texttt{3} \sqsubseteq \texttt{4}$.

Atomic Sections. Our second semantics-preserving transformation is insertion of an atomic section. An atomic section encapsulates a set of statements, and ensures that no concurrent thread can interrupt the execution of these statements.

We represent an atomic section by the set of control locations A it contains. An *atomicity constraint* $\alpha \subseteq 2^{locs(\mathcal{P})}$ is satisfied by a set of atomic sections $\{A_1, A_2, \ldots, A_n\}$ if $\forall A \in \alpha. \exists A_i : A \subseteq A_i$. An atomicity constraint α is weaker than an atomicity constraint α' if $\forall A \in \alpha. \exists A' \in \alpha' : A \subseteq A'$. Any set of atomic sections that satisfies α' also satisfies the weaker α.

Combining Atomicity and Reordering Constraints. A *constraint* is an atomicity- and reordering-constraint pair. Constraint $(\alpha, \sqsubseteq)$ is *weaker* than $(\alpha', \sqsubseteq')$ if either α is weaker than α', or $\alpha = \alpha'$ and $\sqsubseteq$ is weaker than $\sqsubseteq'$. Intuitively, we prefer reordering to inserting atomic sections. We define the conjunction of constraints $(\alpha, \sqsubseteq) \wedge (\alpha', \sqsubseteq')$ as $(\alpha'', \sqsubseteq'')$ where:

- $\alpha'' = \alpha \cup \alpha' \cup \{\{s, s'\} \mid ((s \sqsubseteq s' \wedge s' \sqsubseteq' s) \vee (s \sqsubseteq' s' \wedge s' \sqsubseteq s)) \wedge s \neq s'\}$;
- $\sqsubseteq'' = (\sqsubseteq \cup \sqsubseteq') \setminus \{(s, s') \mid ((s \sqsubseteq s' \wedge s' \sqsubseteq' s) \vee (s \sqsubseteq' s' \wedge s' \sqsubseteq s)) \wedge s \neq s'\}$.

Intuitively, if $\sqsubseteq$ and $\sqsubseteq'$ disagree on the order of s and s', we put them in an atomic section. We define $\top$ as a trivial constraint satisfied by all libraries.

Other Transformations. We motivate another sequential-semantics preserving transformation with an example. Some further transformations are in Section 7.1.

Example 2. In Figure 2, the `timer` thread is invoked when `timer_enabled = 1` to handle requests. The device shutdown thread, `shutdown`, handles the remaining requests and disables the timer. There are two correctness conditions: (1) the timer is disabled after device shutdown; and (2) the `unsafe()` function can accessed only by one thread at a time. Condition (2) is violated as statements `1` and `C` can cause `unsafe` to be executed simultaneously. This happens if `C` calls `unsafe`, and after executing a few instructions of `unsafe`, thread `timer` executes and, in the atomic section, calls `unsafe`. One fix is to move `2` before `1`. This introduces a trace where the assertion fails as the timer gets re-enabled by `1` (switching `1` and `2` is not semantics preserving). A possible fix is to execute statement `2` twice, before and after statement `1`.

```
init: timer_enabled=1, halted=0
timer                          shutdown                      work_queue() {
atomic{                        1: work_queue()                 P: unsafe()
 A: assume(timer_enabled)      2: timer_enabled=0              Q: timer_enabled=1
 B: timer_enabled=0            3: assert(!timer_enabled)     }
 C: work_queue()
}
```

Fig. 2. Example for copying idempotent statements

The above example illustrates another useful semantics-preserving transformation, namely, replication of idempotent statements. A statement s occurring after s' can be replicated before s' if `s; s'; s` has the same effect as `s'; s`.

5 Generalizing Counterexamples to Partial-Order Traces

Partial-order traces. A *po-trace* μ of a concurrent library $\mathcal{P}$ is a tuple $\langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$ where (a) X is a finite set (b) $\preceq$ is a partial-order on X; (c) $loc : X \to Tids \times locs(\mathcal{P})$ is a function labeling X with thread identifiers and control states; and (d) φ_{in} and φ_{end} are state assertions.

A po-trace represents a set of traces of the library. We say a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ is contained in μ if there is a bijection $f : X \to \{0, 1, \ldots, n-1\}$ such that: (a) $f(x) = i \implies loc(x) = (tid, s_i)$ and $\exists \mathcal{V} : (tid, s_i, \mathcal{V}) \in \mathcal{T}_i$; (b) $x_1 \preceq x_2 \implies time_\pi(f(x_1)) \leq time_\pi(f(x_2))$; and (c) $f(x) = 0 \implies (\mathcal{G}_i, \mathcal{T}_i)_{\restriction tid} \models \varphi_{in}) \wedge f(x) = n-1 \implies (\mathcal{G}_n, \mathcal{T}_n)_{\restriction tid} \models \varphi_{end})$. Intuitively, $\pi \in \mu$ if the execution order of statements in π respects the partial order given by $\preceq$, the condition φ_{in} holds at the beginning, and the condition φ_{end} holds at the end. If the order $\preceq$ is linear, we call μ a *linear* po-trace.

As an example, we show how an erroneous trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ such that $\mathcal{G}_n(err) = 1$ is converted into a linear po-trace $\mu_\pi = \langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$. The elements of the tuple are defined as follows. (a) X is {0, 1, ..., n-1 }; (b) $\preceq_\pi$ is the natural order on $\mathbb{N}$; (c) $loc(i) = (tid, s_i)$ if $((\mathcal{G}_i, \mathcal{T}_i), s_i, (\mathcal{G}_{i+1}, \mathcal{T}_{i+1}))$ is a single-step execution of thread tid; and (d) φ_{in} expresses that we start from any initial state; (e) φ_{end} expresses that an error is reached (i.e., it is $err = 1$).

Given two po-traces $\mu = \langle \varphi_{in}, X, \preceq_\mu, loc, \varphi_{end} \rangle$ and $\mu' = \langle \varphi_{in}, X, \preceq_{\mu'}, loc, \varphi_{end} \rangle$ sharing the same trace locations X and loc, and assertions φ_{in} and φ_{end}, we say that μ' is a *relaxation* of μ if $\preceq_\mu \supseteq \preceq_{\mu'}$. Intuitively, a relaxed po-trace puts fewer constraints on the order of execution of statements.

Error invariants. Error invariants were introduced in [6] in a sequential setting. Here we use them to generalize counterexamples to partial-order traces. Let μ be a linear po-trace $\langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$. Without loss of generality, let X be $\{0, 1, \ldots, n-1\}$ and let $\preceq$ be the natural order on $\mathbb{N}$. An *error invariant* *ErrInv* is a function from X to state assertions, such that : (a) $ErrInv(0) = \varphi_{in}$ (b) $ErrInv(i)$ (for $0 < i \leq n-1$) over-approximates the set of states reachable at i along μ. That is, if for a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ we have that if φ_{in} holds for $(\mathcal{G}_0, \mathcal{T}_0)$, then $ErrInv(i)$ holds for $(\mathcal{G}_i, \mathcal{T}_i)$. (c) $ErrInv(i)$ under-approximates the set of states from which we can reach the φ_{end} along μ starting from i. That is, if for a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ we have that if $ErrInv(i)$ holds for $(\mathcal{G}_i, \mathcal{T}_i)$, then φ_{end} holds for $(\mathcal{G}_n, \mathcal{T}_n)$.

As an example, let us consider linear po-trace given by a sequence of statements `A: x=x+1; B:x=2*x; C:y=y+1`, and where φ_{in} is $x = 0 \wedge y = 0$, and φ_{end} is $x > 0 \wedge y > 0$. An error invariant *ErrInv* can be $x > 0 \wedge y \geq 0$ before the execution of `B`, and the same formula before `C`.

We generalize the notion of error invariant to (non-linear) po-traces. Let a po-trace μ be a tuple $\langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$. An error invariant for μ is a function *ErrInv* from X to state assertions such that *ErrInv* is an error invariant for every linear po-trace μ' such that μ is a relaxation of μ'.

5.1 Generalizing Counterexample Traces

We say that a po-trace μ is a counterexample if every trace π contained in μ is erroneous. Given an erroneous trace π, or equivalently, a linear po-trace μ_l, we now present techniques for generalizing it into a non-linear po-trace μ that is a counterexample.

The trace generalization technique proceeds iteratively. Given a po-trace $\mu = \langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$, in each step, we attempt to relax μ by removing the relation $A \preceq B$ for some $A, B \in X$ where A and B correspond to statements from different threads. Further, we require that $\neg \exists P : A \preceq P \preceq B$. However, we need to ensure that the resultant po-trace remains a counterexample after the relaxation, i.e., that every trace contained in it is an erroneous trace. We formalize this condition below.

Let $C, D \in X$ be such that $C \preceq A \preceq B \preceq D$ and $\forall E \in X : E \preceq C \vee C \preceq E \preceq D \vee D \preceq E$. Further, let $\kappa \subseteq X$ be the set $\{E | C \preceq E \preceq D\} \setminus \{D\}$, i.e., κ represents the set of instructions occur between C and D. We call the triple (C, D, κ) a *border set* of A, B, and $\preceq$.

Let J_C and J_D be the error invariants at C and D. Intuitively, we check that we can get from J_C to J_D for every ordering of instructions in κ allowed by $\preceq \setminus (A, B)$. Formally, let X_1, X_2, $\ldots X_n$ be such that each $X_i \in \kappa$ and $\forall E \in \kappa . \exists i : X_i = E$, and $\forall i : X_i \preceq X_{i+1} \vee X_i = B \wedge X_{i+1} = A$. Let s_i be

the instruction corresponding to X_i. We allow relaxing the condition $A \preceq B$ in a step if and only if the following holds: for every sequence $X_1, X_2, \ldots, X_n$ satisfying the above conditions, the Hoare-triple $\{J_C\}s_1; s_2; \ldots; s_n\{J_D\}$ is valid.

Therefore, the full technique for generalizing a trace is as follows: in each step, we pick A and B, and then check the above conditions. If they hold, we relax by removing the pair (A, B) from $\preceq$. Although this technique is sound and complete for generalizing traces, it can be inefficient due to the large number of complex checks needed in each iteration. Instead, we present an alternative algorithm (Algorithm 1) which is sound, but incomplete. The outline of the algorithm is the same as the complete technique presented above, i.e., in each iteration, the algorithm attempts to relax $A \preceq B$. However, we use two alternative checks.

Note that when we try to relax an edge from A to B, we need to check whether this does not invalidate any previous relaxation. Therefore we recheck all the previously relaxed edges in the border set given by A and B.

Algorithm 1. Generalizing linear counterexamples

Input: linear counterexample po-trace μ_l, error invariant *ErrInv*
Output: counterexample po-trace μ that is a relaxation of μ_l

```
1: μ = μ_l
2: for all A ⪯_{μ_l} B do
3:    removeEdge(A, B, μ)
4:    C, D, κ ← borderSet(A, B, μ)
5:    res ← true
6:    for all U, V ∈ κ do
7:       if U ⋠_μ V ∧ V ⋠_μ U then
8:          res ← res ∧ (check1(U, V, C, D, μ, ErrInv) ∨
                         check2(U, V, C, D, μ, ErrInv))
9:    if ¬ res then addEdge(A, B, μ)
10: return μ
```

Rule 1, implemented in procedure *check1*, allows relaxing the order between statements that commute under certain conditions. Let s_U and s_V be the instructions corresponding to U to V. To relax the edge from U to V, we check if there exists K_1 such that $\{J_C\}s_C\{K_1\}$ is a valid Hoare-triple and $K_1 \wedge s_V; s_U \implies K_1 \wedge s_U; s_V$. Intuitively, we are checking if the instructions s_U and s_V commute given the pre-condition K_1. Further, we require that other instructions do not interfere with K_1, i.e., for all $E \in \kappa$ with instruction s_E, K_1 is preserved under s_E, i.e., $\{K_1\}s_E\{K_1\}$ is a valid Hoare-triple.

Rule 2, implemented in procedure *check2*, allows relaxing the order between statements which do not commute, but ensure the similar post-conditions in both orders. The procedure `check2`(U, V, C, D, μ) works as follows. Let J_C be the error invariant at C, and let J_D be the error invariant at D. Let s_U and s_V be the two instructions at nodes U and V. The procedure returns `true` if and only if there exists two state assertions K_1 and K_2 such that all nodes the following conditions hold: (a) $\{J_C\}s_C\{K_1\}$, $\{K_1\}s_U; s_V\{K_2\}$, and $\{K_1\}s_U; s_V\{K_2\}$ are valid Hoare-triples; and (b) $K_2 \rightarrow J_D$. These conditions state that the error

invariants are sufficient to prove that s_u and s_v commute. Furthermore, let E be any other node in κ, and let s_E be the corresponding instruction. We require that s_E preserves K_1 and K_2, i.e., the following two Hoare-triples are valid: (c) $\{K_1\}s_E\{K_1\}$ (d) $\{K_2\}s_E\{K_2\}$ Intuitively, instead of checking all allowed paths from C to D, we find state assertions K_1 and K_2 that are strong enough to prove commutativity, but are preserved by other statements in κ.

Example 3.
- Consider methods `1: x = 0; 2: x = x++` and `A: x = x++; B: assert(x ≤ 1)`. Here, `1` → `A` → `2` → `B` is an erroneous trace. However, the ordering of `A` and `2` is irrelevant to the bug. This order can be eliminated by applying Rule 1 with precondition $K_1 \equiv$ `true`, as we have `A;2` $\Longrightarrow$ `2;A`.
- Using Rule 1 in the illustrative example (Figure 1a) taking K_1 to be $l_1 = 1 \wedge l_2 = 1$ lets us commute the statements `x = l1` and `x = l2`.
- Consider two methods each with the code `1: x = 3` and `A: x = 2; B: assert(x == 0)`. The erroneous trace here is `1` → `A` → `B`. Here, it is clear that `1` and `A` do not commute, i.e., `1;A` $\not\equiv$ `A;1`. However, in the context of this trace, interchanging `A` and `1` still preserves the error. Therefore, using Rule 2 with $K_1 \equiv$ `true` and $K_2 \equiv x > 0$ relax the ordering between `A` and `1`.

We note that Rule 1 and Rule 2 provide only a sound, not a complete proof system for trace generalization. Application of both these rules involve finding suitable K_1 and K_2. The set of conditions imposed on K_1 and K_2 can be expressed as Horn clauses. Solving Horn clauses (in logics useful for program analysis) is a focus of recent research. Non-recursive version was solved by [7], and recursive Horn clauses are solved successfully using heuristics, for example, in [8]. These techniques can be used to implement `check1` and `check2`.

Theorem 1. *Let μ_l be a linear counterexample po-trace corresponding to an erroneous trace, and ErrInv an error invariant for μ_l. If Algorithm 1 returns po-trace μ on μ_l and ErrInv, then μ is a counterexample and a relaxation of μ_l.*

6 Synthesis by Elimination of Partial-Order Counterexamples

We now present Algorithm 2 to solve the synthesis problem stated in Section 3. It works by finding a buggy trace, generalizing it, and then eliminating it using either an atomic section, or a code reordering. The algorithm maintains an atomicity constraint α and a reordering constraint $\sqsubseteq$. In each iteration, library $\mathcal{P}'$ which satisfies $(\alpha, \sqsubseteq)$ is picked and verified. If correct, it is returned. Otherwise, $(\alpha, \sqsubseteq)$ is strengthened using the generalized counterexample. Note that as `Verify` is solving an undecidable problem, it may not terminate. This results in our algorithm not terminating as well. However, as the constraint is strengthened at each step and only a finite number exist, if all calls to `Verify` terminate, then the algorithm terminates and always returns a correct library. This correct library, in the worst case, will have every method enclosed in an atomic section.

Procedure `SemPreservingOrders` was defined in Section 4. `Generalize` is the Algorithm 1. Procedure `Choose` picks a library satisfying a given constraint. `Eliminate` (see below) finds constraints to eliminate a generalized po-trace.

The basic idea behind generalized trace elimination is that $\preceq_\mu$ encodes the happens-before relation among instructions and hence cannot contain loops. Hence, we aim to enforce minimal constraints to introduce a cycle in the $\preceq_\mu$ relation. We extend the graph representing $\preceq_\mu$ by introducing *constraint edges* corresponding to possible atomic sections and reorderings. We then find the smallest cycles, which correspond to the required minimal constraints.

Algorithm 2. Synthesis algorithm

Input: Library $\mathcal{P}$
Output: Error-free library $\mathcal{P}'$ sequentially equivalent to $\mathcal{P}$

```
1: (α, ⊑) ← (∅, SemPreservingOrders(P))
2: while true do
3:    P' ← Choose(⊑, α)
4:    if Verify(P') return P'
5:    μ ← Generalize(cex(P'), ⊑)
6:    (α, ⊑) ← (α, ⊑) ∧ Eliminate(μ, (α, ⊑))
```

Fix a library $\mathcal{P}$ and a partial-order trace $\mu = \langle \varphi_{in}, X, \preceq_\mu, loc, \varphi_{end} \rangle$ for the remainder of this section. The *elimination graph* $G(\mu, \alpha, \sqsubseteq) = (S, E)$ is a weighted graph with vertices $S = X$. The edges $E \subseteq S \times \mathbb{N} \times S$ are described below. Let $x, x' \in S$ and $loc(x) = (tid, s)$ and $loc(x') = (tid', s')$. The function *cons* assigns a constraint to each edge of the elimination graph. We have $(x, w, x') \in E$ if:

- $tid \neq tid' \wedge x \preceq_\mu x' \wedge w = 1 \wedge \neg\exists x'' : x \preceq_\mu x'' \preceq_\mu x'$. In this case, we define $cons((x, w, x')) = \top$.
- $tid = tid'$, where $x \preceq_\mu x'$ and either x and x' belong to different blocks or $s \sqsubseteq s'$. We have $w = |\{x'' \mid x \preceq_\mu x'' \preceq_\mu x'\}|$. Here, we let $cons((x, w, x')) = \top$. These edges correspond to happens-before relations that hold due to $\sqsubseteq$.
- $tid = tid' \wedge s' \sqsubseteq s$ and $w = A \cdot |\{s'' \mid s' \preceq_\mu s'' \preceq_\mu s\}|$ for some constant $A \in \mathbb{N}$. Here, we define $cons((x, w, x')) = (\{s, s'\}, \emptyset)$. Such edges correspond to adding an atomic section around s and s'. We give the atomic section a cost proportional to the minimum number of control locations it contains.
- $tid = tid' \wedge s \not\sqsubseteq s' \wedge s' \not\sqsubseteq s$ and $w = R \cdot |\{(s'', s''') \mid s'' \not\sqsubseteq s''' \wedge s \sqsubseteq s'' \wedge s''' \sqsubseteq s'\}$ for some $R \in \mathbb{N}$. Here, we define $cons((x, w, x')) = (\emptyset, \{(s, s')\})$. This edge corresponds to forcing the order s before s' and has a cost proportional to the number of additional statement orders the constraint implies.

Intuitively, an edge (x, w, x') with $cons((x, w, x')) = \top$ represents a happens-before relation true in any $\mathcal{P}'$ satisfying $(\alpha, \sqsubseteq)$. Every remaining edge (x, w, x') is a happens-before relation true in any library satisfying $cons((x, w, x'))$. We pick A much larger than R to prefer solutions having only reorderings rather than atomic sections (picking A and R such that $A > R \cdot |X|^2$ is sufficient).

Let $x_0 \ldots x_{n-1} x_0$ be a cycle in the elimination graph for a po-trace μ and $(\alpha, \sqsubseteq)$ such that $loc(x_0) = (tid, s) \wedge loc(x_{n-1}) = (tid', s')$ and $tid \neq tid'$. We call

such a cycle an *elimination cycle*. We show that any elimination cycle gives us a constraint that eliminates all traces in po-trace μ. From the elimination cycle, we obtain the following constraint $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$. This is the constraint returned by `Eliminate` (called from Algorithm 2). Fix constraint $(\alpha, \sqsubseteq)$. A constraint $(\alpha', \sqsubseteq')$ *eliminates* a po-trace μ iff all libraries satisfying $(\alpha, \sqsubseteq) \wedge (\alpha, \sqsubseteq)$ and sequentially equivalent to $\mathcal{P}$ do not share a trace with μ.

Theorem 2. *Let $G(\mu, (\alpha, \sqsubseteq))$ contain an elimination cycle $x_0 x_1 \ldots x_{n-1} x_0$. Then, $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$ eliminates the po-trace μ.*

Proof. Say $\pi = (\mathcal{G}_0, \mathcal{T}_0) s_0 (\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1} (\mathcal{G}_n, \mathcal{T}_n) \in \mu$ and let $f : X \to \{1, \ldots, n\}$ be the bijection witnessing the containment. Any trace π in $\mathcal{P}'$ satisfying $(\alpha, \sqsubseteq)$ and $cons(x_i, x_{i+1})$ has $time(f(x_i)) \leq time(f(x_{i+1}))$. Hence, any trace π satisfying $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$ and $(\alpha, \sqsubseteq)$ satisfies $time(f(x_0)) \leq time(f(x_{n-1}))$. However, as (x_{n-1}, x_0) is an edge in the elimination graph where x_0 and x_{n-1} come from different threads, we have that $x_{n-1} \preceq_\pi x_0$ and hence, $time(f(x_{n-1})) \leq time(f(x_0))$. Therefore, we have that $time(f(x_0)) = time(f(x_{n-1}))$. This is not possible as x_0 and x_{n-1} correspond to different threads. Hence, every trace $\pi \in \mu$ is eliminated by $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$. □

Further, the minimal elimination cycle corresponds to a minimal constraint. As $A > R|X|^2$, atomic sections are used iff μ cannot be eliminated by reordering.

Theorem 3. *If $(\alpha, \sqsubseteq)$ is the constraint corresponding to the minimal cycle in the elimination graph, no strictly weaker constraint is sufficient to eliminate μ.*

Finding minimal cycles can be done by running an all-pairs shortest path algorithm, and finding nodes u, v from different threads such that sum of distances u to v and v to u is minimal. Hence, the theorem follows.

Theorem 4. *Finding minimal elimination cycles in the elimination graph $G(\mu, (\alpha, \sqsubseteq))$ can be done in time polynomial in the size of μ, α, and $\sqsubseteq$.*

7 Application to Systems Code

7.1 A Study of Concurrency Bugs in Linux Drivers

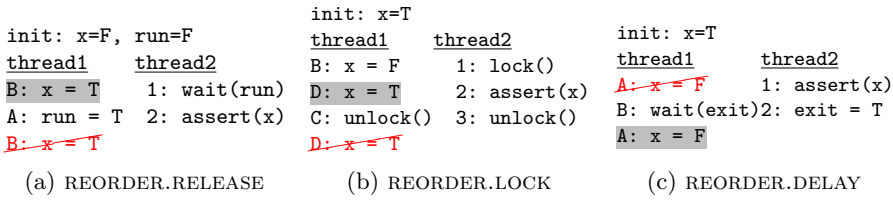
Our work is motivated by a study of concurrency defects in Linux device drivers. Drivers are required to perform well under concurrent workloads, which calls for sparing and fine-grained use of locks. This, in turn, provokes many concurrency-related bugs, making concurrency a major source of errors in drivers [4,12]. Our study considered 100 most recent (as of Dec. 2012) concurrency-related defects fixed in Linux device drivers (we used the Linux kernel development archive obtained from `www.kernel.org`). These defects occurred in 68 different drivers, all maintained by different developers. For each bug, we rely on manual code inspection to understand the exact nature of the bug and the fix.

We observed that many bug fixes involve subtle and seemingly ad hoc code transformations. In-depth analysis reveals several common patterns, shown in

Table 1. Synchronization patterns in Linux device drivers

pattern	description	#
REORDER	Reorder program statements to eliminate a race	28
LOCK	Protect racing code sections with a lock	17
OPTIMISTIC	Check if another thread has modified the value of a shared variable	10
BARRIER	Use a system-provided function to wait for a racing thread to terminate or complete a critical section	7
ATOMIC	Replace a statement-sequence with an equivalent atomic primitive	6
UPGRADE	Replace a synchronization primitive with a stronger one	5
UNSHARE	Avoid sharing by creating a private copy of a shared variable	3
CLONE	Replicate an idempotent statement	1
ADHOC	Transformations that do not fall into one of the previous categories	23
Total		**100**

Table 1. In particular, 28 of 100 fixes were semantic-preserving statement reorderings (the REORDER pattern). These further fall into several subpatterns (see Table 2 and Figure 3). Reordering instructions often involves additional side effects. For example, moving a statement across function boundaries may require adding arguments or return values to functions. Our implementation currently does not perform these, but can be extended to do so.



(a) REORDER.RELEASE (b) REORDER.LOCK (c) REORDER.DELAY

Fig. 3. Examples of REORDER subpatterns and corresponding elimination graphs

Interestingly, LOCK pattern (17%) is rarer than expected. Performance and kernel-imposed constraints often prevent lock usage. This observation confirms that locks are not a universal band-aid for concurrency defects in OS code. We do not discuss remaining bug categories, but note that we encountered 23 bug fixes that did not fit into any pattern (AD HOC in Table 1). We expect to discover new patterns among these as we include more defects in our study.

7.2 Synthesis Case Study

We implemented our algorithms in a tool called ConcurrencySwapper (Source and benchmarks can be found here: `https://github.com/thorstent/ConcurrencySwapper`). It handles a restricted subset of C, avoiding complex parts including pointer arithmetic, aliasing, bit-wise arithmetic, etc. It uses CPAChecker [2] to convert C statements into formulae representing instructions, as in Section 3. We use the bounded model checking tool Poirot [1] to detect three kinds of bugs: (a) assertion failures; (b) generic correctness

Table 2. Subpatterns of the REORDER pattern

pattern	description	example	#
REORDER.RELEASE	Move a variable assignment to a location before another thread accessing this variable is released	Fig 3a	11
REORDER.LOCK	Move statements to existing lock-protected section	Fig 3b	10
REORDER.DELAY	Delay assignment to a shared variable until a racing thread accessing this variable has terminated	Fig 3c	6
REORDER.RW	Reorder accesses to a pair of shared variables	Fig 1b	1
REORDER.ADHOC	Application-specific reordering	–	1

conditions (e.g., initialization-before-use for pointers); and (c) deadlocks (as Poirot does not detect deadlocks, we manually encoded these as suitable assertions for our examples). We generalize buggy traces, using Z3 theorem prover [5] to perform the required checks for Rules 1 and 2. The current implementation does not compute invariants during generalization; but even without invariant computation, our tool came up with the right program transformations quickly. To evaluate the effectiveness of trace generalization, we ran the experiments with and without it.

Reporting. Although each iteration of the algorithm eliminates a buggy po-trace, additional traces may exhibit the same bug. We report the iterations needed to completely fix a bug, i.e., until no more traces exhibit a similar bug. Also, we report separately, the time taken to: (a) find bugs; (b) generalize the trace and find a fix; and (c) verify the correct program. We report the verification time separately as it is usually the largest fraction of execution time.

Benchmarks. Our initial evaluation consisted of 5 microbenchmarks each of 15–30 lines of code without comments, and modeling a single concurrency defect found in a real Linux driver. The iterations required and fix patterns are summarized in Table 3. The synthesis took less than 15 seconds for each case, with trace analysis taking less than 0.5 seconds. Also, in 1 case, not using trace generalization leads to an additional iteration, leading to a larger execution time.

Table 3. Micro-benchmarks

Benchmark	Fix pattern	Iters.	Iters (w/o trace gen.)
ex1	REORDER.RW	1	1
ex2	REORDER.RELEASE	1	1
ex3	REORDER.LOCK	1	1
ex4	REORDER.ADHOC	3	3
ex5	LOCK	2	3

We evaluate the scalability of ConcurrencySwapper using a simplified version of the Linux Realtek 8169 driver. This driver is representative of medium to high-end drivers both in terms of overall complexity and the complexity of synchronization logic. We extracted the driver's complete synchronization skeleton, including code and variables related to thread synchronization and communication. The skeleton does not include the actual device management code, which is irrelevant to concurrency, and was additionally simplified to avoid currently unsupported C constructs. We provide an environment model to simulate all (7) OS threads that interact with the driver. The

Table 4. Results for Linux Realtek 8169 driver benchmark

Bug	Fix pattern	With trace generalization		Without trace generalization	
		Iters.	Bug-finding	Iters.	Additional bug-finding
bug1	REORDER.RELEASE	1	8 sec	1	same
bug2	REORDER.DELAY	1	23 sec	4	same + 80 sec
bug3	REORDER.RW	1	93 sec	1	same
bug4	REORDER.RW	1	94 sec	1	same
bug5	REORDER.ADHOC	2	47 sec	2	same

resulting skeleton had 364 LOC, while the original driver had around 7,000 LOC. The skeleton had 5 concurrency defects.

Poirot was able to find all the defects, and ConcurrencySwapper was able to find fixes for each defect through statement reordering. The results are summarized in Table 4. In each iteration, the trace analysis phase took less than 2 seconds. The extra bug finding times due to additional iterations is reported for the runs without trace generalization. In one case, the 3 additional iterations were required without trace generalization. The bug finding times dominate the trace analysis times, justifying the use of complex trace generalization procedure to avoid additional iterations. The verification phase took around 30 minutes.

8 Conclusion

The contributions of the paper are two-fold. First, our synthesis procedure considers a variety of semantics-preserving transformations (not just lock placement). Second, in order to speed up the synthesis procedure, we consider counterexamples that are partial orders on instructions (as opposed to linear orders). There are several possible directions for future work. First, we will investigate generalizations of counterexamples in CEGAR algorithms for verification, rather than synthesis. Second, we plan to address issues in systems programming such as weak memory models. In this paper, we compute minimal sufficient ordering constraints. In a sequentially-consistent models, reordering statements alone is sufficient to enforce the constraints; but in a weak model, additional fences may be needed to enforce them.

References

1. Poirot: The concurrency sleuth, `http://research.microsoft.com/en-us/projects/poirot/`
2. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
3. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI, pp. 304–315 (2008)
4. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: SOSP, pp. 73–88 (2001)

5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012)
7. Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 188–203. Springer, Heidelberg (2011)
8. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
9. Lipton, R.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
10. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 278–324. Springer, Heidelberg (1987)
11. Nielson, F., Nielson, H., Hankin, C.: Principles of program analysis (2. corr. print). Springer (2005)
12. Ryzhyk, L., Chubb, P., Kuz, I., Heiser, G.: Dingo: Taming device drivers. In: Eurosys (2009)
13. Solar-Lezama, A., Jones, C., Bodík, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148 (2008)
14. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI, pp. 125–135 (2008)
15. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)