From Non-preemptive to Preemptive Scheduling using Synchronization Synthesis *

Pavol Černý[1], Edmund M. Clarke[2], Thomas A. Henzinger[3], Arjun Radhakrishna[4], Leonid Ryzhyk[2], Roopsha Samanta[3], and Thorsten Tarrach[3]

[1] University of Colorado Boulder
[2] Carnegie Mellon University
[3] IST Austria
[4] University of Pennsylvania

Abstract. We present a computer-aided programming approach to concurrency. The approach allows programmers to program assuming a friendly, non-preemptive scheduler, and our synthesis procedure inserts synchronization to ensure that the final program works even with a preemptive scheduler. The correctness specification is implicit, inferred from the non-preemptive behavior. Let us consider sequences of calls that the program makes to an external interface. The specification requires that any such sequence produced under a preemptive scheduler should be included in the set of such sequences produced under a non-preemptive scheduler. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and rules for inserting synchronization. We apply the approach to device-driver programming, where the driver threads call the software interface of the device and the API provided by the operating system. Our experiments demonstrate that our synthesis method is precise and efficient, and, since it does not require explicit specifications, is more practical than the conventional approach based on user-provided assertions.

1 Introduction

Concurrent shared-memory programming is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [4, 5, 8, 11]. However, specifying the programmer's intent may be a challenge in itself. Declarative mechanisms, such as assertions, suffer from the drawback that it is difficult to ensure that the specification is complete and fully captures the programmer's intent.

We propose a solution where the specification is *implicit*. We observe that a core difficulty in concurrent programming originates from the fact that the scheduler can *preempt* the execution of a thread at any time. We therefore give

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM), by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award), by NSF under award CCF 1421752 and the Expeditions award CCF 1138996, by the Simons Foundation, and by a gift from the Intel Corporation.

the developer the option to program assuming a friendly, *non-preemptive*, scheduler. Our tool automatically synthesizes synchronization code to ensure that every behavior of the program under preemptive scheduling is included in the set of behaviors produced under non-preemptive scheduling. Thus, we use the non-preemptive semantics as an implicit correctness specification.

The non-preemptive scheduling model dramatically simplifies the development of concurrent software, including operating system (OS) kernels, network servers, database systems, etc. [13, 14]. In this model, a thread can only be descheduled by voluntarily yielding control, e.g., by invoking a blocking operation. Synchronization primitives may be used for communication between threads, e.g., a producer thread may use a semaphore to notify the consumer about availability of data. However, one does not need to worry about protecting accesses to shared state: a series of memory accesses executes atomically as long as the scheduled thread does not yield.

In defining behavioral equivalence between preemptive and non-preemptive executions, we focus on externally observable program behaviors: two program executions are *observationally equivalent* if they generate the same sequences of calls to interfaces of interest. This approach facilitates modular synthesis where a module's behavior is characterized in terms of its interaction with other modules. Given a multi-threaded program $\mathcal{C}$ and a synthesized program $\mathcal{C}'$ obtained by adding synchronization to $\mathcal{C}$, $\mathcal{C}'$ is *preemption-safe* w.r.t. $\mathcal{C}$ if for each execution of $\mathcal{C}'$ under a preemptive scheduler, there is an observationally equivalent non-preemptive execution of $\mathcal{C}$. Our synthesis goal is to automatically generate a preemption-safe version of the input program.

We rely on abstraction to achieve efficient synthesis of multi-threaded programs. We propose a simple, *data-oblivious* abstraction inspired by an analysis of synchronization patterns in OS code, which tend to be independent of data values. The abstraction tracks types of accesses (read or write) to each memory location while ignoring their values. In addition, the abstraction tracks branching choices. Calls to an external interface are modeled as writes to a special memory location, with independent interfaces modeled as separate locations. To the best of our knowledge, our proposed abstraction is yet to be explored in the verification and synthesis literature.

Two abstract program executions are observationally equivalent if they are equal modulo the classical independence relation I on memory accesses: accesses to different locations are independent, and accesses to the same location are independent iff they are both read accesses. Using this notion of equivalence, the notion of preemption-safety is extended to abstract programs.

Under abstraction, we model each thread as a nondeterministic finite automaton (NFA) over a finite alphabet, with each symbol corresponding to a read or a write to a particular variable. This enables us to construct NFAs N, representing the abstraction of the original program $\mathcal{C}$ under non-premptive scheduling, and P, representing the abstraction of the synthesized program $\mathcal{C}'$ under preemptive scheduling. We show that preemption-safety of $\mathcal{C}'$ w.r.t. $\mathcal{C}$ is implied by preemption-safety of the abstract synthesized program w.r.t. the abstract original program, which, in turn, is implied by language inclusion modulo I of NFAs P and N. While the problem of language inclusion modulo an indepen-

dence relation is undecidable [2], we show that the antichain-based algorithm for standard language inclusion [9] can be adapted to decide a bounded version of language inclusion modulo an independence relation.

Our overall synthesis procedure works as follows: we run the algorithm for bounded language inclusion modulo I, iteratively increasing the bound, until it reports that the inclusion holds, or finds a counterexample, or reaches a timeout. In the first case, the synthesis procedure terminates successfully. In the second case, the counterexample is generalized to a set of counterexamples represented as a Boolean combination of ordering constraints over control-flow locations (as in [11]). These constraints are analyzed for patterns indicating the type of concurrency bug (atomicity, ordering violation) and the type of applicable fix (lock insertion, statement reordering). After applying the fix(es), the procedure is restarted from scratch; the process continues until we find a preemption-safe program, or reach a timeout.

We implemented our synthesis procedure in a new prototype tool called Liss (Language Inclusion-based Synchronization Synthesis) and evaluated it on a series of device driver benchmarks, including an Ethernet driver for Linux and the synchronization skeleton of a USB-to-serial controller driver. First, Liss was able to detect and eliminate all but two known race conditions in our examples; these included one race condition that we previously missed when synthesizing from explicit specifications [5], due to a missing assertion. Second, our abstraction proved highly efficient: Liss runs an order of magnitude faster on the more complicated examples than our previous synthesis tool based on the CBMC model checker. Third, our coarse abstraction proved surprisingly precise in practice: across all our benchmarks, we only encountered three program locations where manual abstraction refinement was needed to avoid the generation of unnecessary synchronization. Overall, our evaluation strongly supports the use of the implicit specification approach based on non-preemptive scheduling semantics as well as the use of the data-oblivious abstraction to achieve practical synthesis for real-world systems code.

Contributions. First, we propose a new specification-free approach to synchronization synthesis. Given a program written assuming a friendly, non-preemptive scheduler, we automatically generate a preemption-safe version of the program. Second, we introduce a novel abstraction scheme and use it to reduce preemption-safety to language inclusion modulo an independence relation. Third, we present the first language inclusion-based synchronization synthesis procedure and tool for concurrent programs. Our synthesis procedure includes a new algorithm for a bounded version of our inherently undecidable language inclusion problem. Finally, we evaluate our synthesis procedure on several examples. To the best of our knowledge, Liss is the first synthesis tool capable of handling realistic (albeit simplified) device driver code, while previous tools were evaluated on small fragments of driver code or on manually extracted synchronization skeletons.

Related work. Synthesis of synchronization is an active research area [3–6, 10–12, 15, 16]. Closest to our work is a recent paper by Bloem et al. [3], which uses implicit specifications for synchronization synthesis. While their specification is given by sequential behaviors, ours is given by non-preemptive behaviors. This makes our approach applicable to scenarios where threads need to communicate

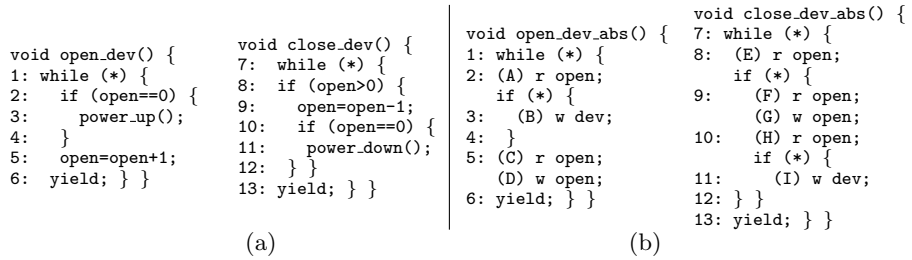

Fig. 1: Running example and its abstraction

explicitly. Further, correctness in [3] is determined by comparing values at the end of the execution. In contrast, we compare sequences of events, which serves as a more suitable specification for infinitely-looping reactive systems.

Many efforts in synthesis of synchronization focus on user-provided specifications, such as assertions (our previous work [4, 5, 11]). However, it is hard to determine if a given set of assertions represents a complete specification. In this paper, we are solving language inclusion, a computationally harder problem than reachability. However, due to our abstraction, our tool performs significantly better than tools from [4,5], which are based on a mature model checker (CBMC [7]). Our abstraction is reminiscent of previously used abstractions that track reads and writes to individual locations (e.g., [1,17]). However, our abstraction is novel as it additionally tracks some control-flow information (specifically, the branches taken) giving us higher precision with almost negligible computational cost. The synthesis part of our approach is based on [11].

In [16] the authors rely on assertions for synchronization synthesis and include iterative abstraction refinement in their framework. This is an interesting extension to pursue for our abstraction. In other related work, CFix [12] can detect and fix concurrency bugs by identifying simple bug patterns in the code.

2 Illustrative Example

Fig. 1a contains our running example. Consider the case where the procedures `open_dev()` and `close_dev()` are invoked in parallel, possibly multiple times (modeled as a non-deterministic while loop). The functions `power_up()` and `power_down()` represent calls to a device. For the non-preemptive scheduler, the sequence of calls to the device will always be a repeating sequence of one call to `power_up()`, followed by one call to `power_down()`. Without additional synchronization, however, there could be two calls to `power_up()` in a row when executing it with a preemptive scheduler. Such a sequence is not observationally equivalent to any sequence that can be produced when executing with a non-preemptive scheduler.

Fig. 1b contains the abstracted versions (we omit tracking of branching choices in the example)of the two procedures, `open_dev_abs()` and `close_dev_abs()`. For instance, the instruction `open = open + 1` is abstracted to the two instructions labeled (C) and (D). The abstraction is coarse, but still captures the problem. Consider two threads `T1` and `T2` running the `open_dev_abs()` procedure. The following trace is possible under a preemptive scheduler, but not under a non-preemptive scheduler: `T1.A; T2.A; T1.B;`

`T1.C; T1.D; T2.B; T2.C; T2.D`. Moreover, the trace cannot be transformed by swapping independent events into any trace possible under a non-preemptive scheduler. This is because instructions `A` and `D` are not independent. Hence, the abstract trace exhibits the problem of two successive calls to `power_up()` when executing with a preemptive scheduler. Our synthesis procedure finds this problem, and fixes it by introducing a lock in `open_dev()` (see Sec. 5).

3 Preliminaries and Problem Statement

Syntax. We assume that programs are written in a concurrent while language $\mathcal{W}$. A concurrent program $\mathcal{C}$ in $\mathcal{W}$ is a finite collection of threads $\langle \mathtt{T}_1, \ldots, \mathtt{T}_n \rangle$ where each thread is a statement written in the syntax from Fig. 2. All $\mathcal{W}$ variables (program variables `std_var`, lock variables `lock_var`, and condition variable `cond_var`) range over integers and each statement is labeled with a unique location identifier l. The only non-standard syntactic constructs in $\mathcal{W}$ relate to the *tags*. Intuitively, each tag is a communication channel between the program and an interface to an external system, and the $\mathsf{input}(\mathsf{tag})$ and $\mathsf{output}(\mathsf{tag}, \mathsf{expr})$ statements read from and write to the channel. We assume that the program and the external system interface can only communicate through the channel. In practice, we use the tags to model device registers. In our presentation, we consider only a single external interface. Our implementation can handle communication with several interfaces.

```
expr ::= std_var | constant | operator(expr, expr, ..., expr)
lstmt ::= loc: stmt | lstmt; lstmt
stmt ::= skip | std_var := expr | std_var := havoc()
 | if (expr) lstmt else lstmt | while (expr) lstmt | std_var := input(tag)
 | output(tag, expr) | lock(lock_var)  | unlock(lock_var)
 | signal(cond_var)  | await(cond_var) | reset(cond_var) | yield
```

Fig. 2: Syntax of $\mathcal{W}$

Semantics. We begin by defining the semantics of a single thread in $\mathcal{W}$, and then extend the definition to concurrent non-preemptive and preemptive semantics. Note that in our work, reads and writes are assumed to execute atomically and further, we assume a sequentially consistent memory model.

Single-thread semantics. A program state is given by $\langle \mathcal{V}, P \rangle$ where $\mathcal{V}$ is a valuation of all program variables, and P is the statement that remains to be executed. Let us fix a thread identifier *tid*.

The operational semantics of a thread executing in isolation is given in Fig. 3. A single execution step $\langle \mathcal{V}, P \rangle \xrightarrow{\alpha} \langle \mathcal{V}', P' \rangle$ changes the program state from $\langle \mathcal{V}, P \rangle$ to $\langle \mathcal{V}', P' \rangle$ while optionally outputting an *observable symbol* α. The absence of a symbol is denoted using ϵ. Most rules from Fig. 3 are standard—the special rules are the Havoc, Input, and Output rules.

1. Havoc: Statement $l : x := \mathsf{havoc}$ assigns x a non-deterministic value (say k) and outputs the observable $(tid, \mathsf{havoc}, k, x)$.
2. Input, Output: $l : x := \mathsf{input}(t)$ and $l : \mathsf{output}(t, e)$ read and write values to the channel t, and output $(tid, \mathsf{input}, k, t)$ and $(tid, \mathsf{output}, k, t)$, where k is the value read or written, respectively.

Intuitively, the observables record the sequence of non-deterministic guesses, as well as the input/output interaction with the tagged channels. In the following, e represents an expression and $e[v/\mathcal{V}[v]]$ evaluates an expression by replacing all variables v with their values in $\mathcal{V}$.

$$\frac{e[v/\mathcal{V}[v]] = k}{\langle \mathcal{V}, l: \ x := e\rangle \xrightarrow{\epsilon} \langle \mathcal{V}[x := k], \mathsf{skip}\rangle}\textsc{Assign} \qquad \frac{k \in \mathbb{N} \qquad \alpha = (tid, \mathsf{havoc}, k, x)}{\langle \mathcal{V}, l: \ x := \mathsf{havoc}\rangle \xrightarrow{\alpha} \langle \mathcal{V}[x := k], \mathsf{skip}\rangle}\textsc{Havoc}$$

$$\frac{e[v/\mathcal{V}[v]] = \mathsf{false}}{\langle \mathcal{V}, l: \ \mathsf{while}(e)\ s\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, \mathsf{skip}\rangle}\textsc{While1} \qquad \frac{e[v/\mathcal{V}[v]] = \mathsf{true}}{\langle \mathcal{V}, l: \ \mathsf{while}(e)\ s\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, s; \mathsf{while}(e)\ s\rangle}\textsc{While2}$$

$$\frac{e[v/\mathcal{V}[v]] = \mathsf{true}}{\langle \mathcal{V}, l: \ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, s_1\rangle}\textsc{If1} \qquad \frac{e[v/\mathcal{V}[v]] = \mathsf{false}}{\langle \mathcal{V}, l: \ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, s_2\rangle}\textsc{If2}$$

$$\frac{\langle \mathcal{V}, s_1\rangle \xrightarrow{\alpha} \langle \mathcal{V}', s_1'\rangle}{\langle \mathcal{V}, l: \ s_1; s_2\rangle \xrightarrow{\alpha} \langle \mathcal{V}', s_1'; s_2\rangle}\textsc{Sequence} \qquad \frac{k \in \mathbb{N} \qquad \alpha = (tid, \mathsf{input}, k, t)}{\langle \mathcal{V}, l: \ x := \mathsf{input}(t)\rangle \xrightarrow{\alpha} \langle \mathcal{V}[x := k], \mathsf{skip}\rangle}\textsc{Input}$$

$$\frac{}{\langle \mathcal{V}, l: \ \mathsf{skip}; s_2\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, s_2\rangle}\textsc{Skip} \qquad \frac{e[v/\mathcal{V}[v]] = k \qquad \alpha = (tid, \mathsf{output}, k, t)}{\langle \mathcal{V}, l: \ \mathsf{output}(t, e)\rangle \xrightarrow{\alpha} \langle \mathcal{V}, \mathsf{skip}\rangle}\textsc{Output}$$

Fig. 3: Single thread semantics of $\mathcal{W}$

Non-preemptive semantics. The non-preemptive semantics of $\mathcal{W}$ is presented in Appendix A. The non-preemptive semantics ensures that a single thread from the program keeps executing as detailed above until one of the following occurs: (a) the thread finishes execution, or it encounters (b) a yield statement, or (c) a lock statement and the lock is taken, or (d) an await statement and the condition variable is not set. In these cases, a context-switch is possible.
Preemptive semantics. The preemptive semantics of a program is obtained from the non-preemptive semantics by relaxing the condition on context-switches, and allowing context-switches at all program points (see Appendix A).

3.1 Problem statement

A *non-preemptive observation sequence* of a program $\mathcal{C}$ is a sequence $\alpha_0 \dots \alpha_k$ if there exist program states S_0^{pre}, S_0^{post}, $\dots$, S_k^{pre}, S_k^{post} such that according to the non-preemptive semantics of $\mathcal{W}$, we have: (a) for each $0 \le i \le k$, $\langle S_i^{pre}\rangle \xrightarrow{\alpha_i} \langle S_i^{post}\rangle$, (b) for each $0 \le i < k$, $\langle S_i^{post}\rangle \xrightarrow{\epsilon}{}^* \langle S_{i+1}^{pre}\rangle$, and (c) for the initial state S_ι and a final state (i.e., where all threads have finished execution) S_f, $\langle S_\iota\rangle \xrightarrow{\epsilon}{}^* \langle S_0^{pre}\rangle$ and $\langle S_k^{post}\rangle \xrightarrow{\epsilon}{}^* \langle S_f\rangle$. Similarly, a *preemptive observation sequence* of a program $\mathcal{C}$ is a sequence $\alpha_0 \dots \alpha_k$ as above, with the non-preemptive semantics replaced with preemptive semantics. We denote the sets of non-preemptive and preemptive observation sequences of a program $\mathcal{C}$ by $[\![\mathcal{C}]\!]^{NP}$ and $[\![\mathcal{C}]\!]^{P}$, respectively.

We say that observation sequences $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ are *equivalent* if:
- The subsequences of $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ containing only symbols of the form $(tid, \mathsf{Input}, k, t)$ and $(tid, \mathsf{Output}, k, t)$ are equal, and
- For each thread identifier tid, the subsequences of $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ containing only symbols of the form $(tid, \mathsf{Havoc}, k, x)$ are equal.

Intuitively, observable sequences are equivalent if they have the same interaction with the interface, and the same non-deterministic choices in each thread. For

sets of observable sequences $\mathcal{O}_1$ and $\mathcal{O}_2$, we write $\mathcal{O}_1 \subseteq \mathcal{O}_2$ to denote that each sequence in $\mathcal{O}_1$ has an equivalent sequence in $\mathcal{O}_2$. Given a concurrent program $\mathcal{C}$ and a synthesized program $\mathcal{C}'$ obtained by adding synchronization to $\mathcal{C}$, the program $\mathcal{C}'$ is *preemption-safe* w.r.t. $\mathcal{C}$ if $[\![\mathcal{C}']\!]^P \subseteq [\![\mathcal{C}]\!]^{NP}$.

We are now ready to state our synthesis problem. Given a concurrent program $\mathcal{C}$, the aim is to synthesize a program $\mathcal{C}'$, by adding synchronization to $\mathcal{C}$, such that $\mathcal{C}'$ is preemption-safe w.r.t. $\mathcal{C}$.

3.2 Language Inclusion Modulo an Independence Relation

We reduce the problem of checking if a synthesized solution is preemption-safe w.r.t. the original program to an automata-theoretic problem.

Abstract semantics for $\mathcal{W}$. We first define a single-thread abstract semantics for $\mathcal{W}$ (Fig. 4), which tracks types of accesses (read or write) to each memory location while abstracting away their values. Inputs/outputs to an external interface are modeled as writes to a special memory location (`dev`). Even inputs are modeled as writes because in our applications we cannot assume that reads from the external interface are free of side-effects. Havocs become ordinary writes to the variable they are assigned to. Every branch is taken non-deterministically and tracked. The only constructs preserved are the lock and condition variables. The abstract program state consists of the valuations of the lock and condition variables and the statement that remains to be executed. In the abstraction, an observable is of the form $(tid, \{\mathsf{read}, \mathsf{write}, \mathsf{exit}, \mathsf{loop}, \mathsf{then}, \mathsf{else}\}, v, l)$ and observes the type of access (read/write) to variable v and records non-deterministic branching choices (exit/loop/then/else). The latter are not associated with any variable.

In Fig. 4, given expression e, the function $\mathit{Reads}(tid, e, l)$ represents the sequence $(tid, \mathsf{read}, v_1, l) \cdot \ldots \cdot (tid, \mathsf{read}, v_n, l)$ where $v_1, \ldots, v_n$ are the variables in e, in the order they are read to evaluate e.

$$\frac{\alpha = \mathit{Reads}(tid, e, l) \cdot (tid, \mathsf{write}, x, l)}{\langle \mathcal{V}, l : \ x := e \rangle \xrightarrow{\alpha} \langle \mathcal{V}, \mathsf{skip} \rangle}\textsc{Assign} \qquad \frac{\alpha = (tid, \mathsf{write}, x, l)}{\langle \mathcal{V}, l : \ x := \mathsf{havoc} \rangle \xrightarrow{\alpha} \langle \mathcal{V}, \mathsf{skip} \rangle}\textsc{Havoc}$$

$$\frac{\alpha = \mathit{Reads}(tid, e, l) \cdot (tid, \mathsf{exit}, \_, l)}{\langle \mathcal{V}, l : \ \mathsf{while}(e)\ s \rangle \xrightarrow{\alpha} \langle \mathcal{V}, \mathsf{skip} \rangle}\textsc{While1} \qquad \frac{\alpha = \mathit{Reads}(tid, e, l) \cdot (tid, \mathsf{loop}, \_, l)}{\langle \mathcal{V}, l : \ \mathsf{while}(e)\ s \rangle \xrightarrow{\alpha} \langle \mathcal{V}, s; \mathsf{while}(e)\ s \rangle}\textsc{While2}$$

$$\frac{\alpha = \mathit{Reads}(tid, e, l) \cdot (tid, \mathsf{then}, \_, l)}{\langle \mathcal{V}, l : \ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rangle \xrightarrow{\alpha} \langle \mathcal{V}, s_1 \rangle}\textsc{If1} \qquad \frac{\alpha = \mathit{Reads}(tid, e, l) \cdot (tid, \mathsf{else}, \_, l)}{\langle \mathcal{V}, l : \ \mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rangle \xrightarrow{\alpha} \langle \mathcal{V}, s_2 \rangle}\textsc{If2}$$

$$\frac{\langle \mathcal{V}, s_1 \rangle \xrightarrow{\alpha} \langle \mathcal{V}', s_1' \rangle}{\langle \mathcal{V}, l : \ s_1; s_2 \rangle \xrightarrow{\alpha} \langle \mathcal{V}', s_1'; s_2 \rangle}\textsc{Sequence} \qquad \frac{\alpha = (tid, \mathsf{write}, \mathsf{dev}, l) \cdot (tid, \mathsf{write}, x, l)}{\langle l : \ x := \mathsf{input}(t) \rangle \xrightarrow{\alpha} \langle \mathsf{skip} \rangle}\textsc{Input}$$

$$\frac{}{\langle \mathcal{V}, l : \ \mathsf{skip}; s_2 \rangle \xrightarrow{\epsilon} \langle \mathcal{V}, s_2 \rangle}\textsc{Skip} \qquad \frac{\alpha = \mathit{Reads}(tid, e, l) \cdot (tid, \mathsf{write}, \mathsf{dev}, l)}{\langle \mathcal{V}, l : \ \mathsf{output}(t, e) \rangle \xrightarrow{\alpha} \langle \mathcal{V}, \mathsf{skip} \rangle}\textsc{Output}$$

Fig. 4: Single thread abstract semantics of $\mathcal{W}$

The abstract program semantics (Figures 6 and 7) is the same as the concrete program semantics where the single thread semantics is replaced by the abstract

single thread semantics. Locks and conditionals and operations on them are not abstracted.

As with the concrete semantics of $\mathcal{W}$, we can define the non-preemptive and preemptive observable sequences for abstract semantics. For a concurrent program $\mathcal{C}$, we denote the sets of abstract preemptive and non-preemptive observable sequences by $[\![\mathcal{C}]\!]^P_{abs}$ and $[\![\mathcal{C}]\!]^{NP}_{abs}$, respectively.

Abstract observation sequences $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ are *equivalent* if:

- For each thread tid, the subsequences of $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ containing only symbols of the form (tid, a, v, l), with $a \in \{\mathsf{read}, \mathsf{write}, \mathsf{exit}, \mathsf{loop}, \mathsf{then}, \mathsf{else}\}$ are equal,
- For each variable v, the subsequences of $\alpha_0 \dots \alpha_k$ and $\beta_0 \dots \beta_k$ containing only write symbols (of the form $(tid, \mathsf{write}, v, l)$) are equal, and
- For each variable v, the multisets of symbols of the form $(tid, \mathsf{read}, v, l)$ between any two write symbols, as well as before the first write symbol and after the last write symbol are identical.

We first show that the abstract semantics is sound w.r.t. preemption-safety (see Appendix B for the proof).

Theorem 1. *Given concurrent program $\mathcal{C}$ and a synthesized program $\mathcal{C}'$ obtained by adding synchronization to $\mathcal{C}$, $[\![\mathcal{C}']\!]^P_{abs} \subseteq [\![\mathcal{C}]\!]^{NP}_{abs} \Rightarrow [\![\mathcal{C}']\!]^P \subseteq [\![\mathcal{C}]\!]^{NP}$.*

Abstract semantics to automata. An NFA $\mathcal{A}$ is a tuple $(Q, \Sigma, \Delta, Q_\iota, F)$ where Σ is a finite alphabet, Q, Q_ι, F are finite sets of states, initial states and final states, respectively and Δ is a set of transitions. A word $\sigma_0 \dots \sigma_k \in \Sigma^*$ is *accepted* by $\mathcal{A}$ if there exists a sequence of states $q_0 \dots q_{k+1}$ such that $q_0 \in Q_\iota$ and $q_{k+1} \in F$ and $\forall i : (q_i, \sigma_i, q_{i+1}) \in \Delta$. The set of all words accepted by $\mathcal{A}$ is called the language of $\mathcal{A}$ and is denoted $\mathcal{L}(\mathcal{A})$.

Given a program $\mathcal{C}$, we can construct automata $\mathcal{A}([\![\mathcal{C}]\!]^{NP}_{abs})$ and $\mathcal{A}([\![\mathcal{C}]\!]^{P}_{abs})$ that accept exactly the observable sequences under the respective semantics. We describe their construction informally. Each automaton state is a program state of the abstract semantics and the alphabet is the set of abstract observable symbols. There is a transition from one state to another on an observable symbol (or an ϵ) iff the program can execute one step under the corresponding semantics to reach the other state while outputting the observable symbol (on an ϵ).

Language inclusion modulo an independence relation. Let I be a non-reflexive, symmetric binary relation over an alphabet Σ. We refer to I as the *independence relation* and to elements of I as *independent* symbol pairs. We define a symmetric binary relation $\approx$ over words in Σ^*: for all words $\sigma, \sigma' \in \Sigma^*$ and $(\alpha, \beta) \in I$, $(\sigma \cdot \alpha\beta \cdot \sigma', \sigma \cdot \beta\alpha \cdot \sigma') \in \approx$. Let $\approx^t$ denote the reflexive transitive closure of $\approx$.[5] Given a language $\mathcal{L}$ over Σ, the closure of $\mathcal{L}$ w.r.t. I, denoted $\mathrm{Clo}_I(\mathcal{L})$, is the set $\{\sigma \in \Sigma^* : \exists \sigma' \in \mathcal{L}$ with $(\sigma, \sigma') \in \approx^t\}$. Thus, $\mathrm{Clo}_I(\mathcal{L})$ consists of all words that can be obtained from some word in $\mathcal{L}$ by repeatedly commuting adjacent independent symbol pairs from I.

Definition 1 (Language inclusion modulo an independence relation). *Given NFAs A, B over a common alphabet Σ and an independence relation I over Σ, the language inclusion problem modulo I is: $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$?*

[5] The equivalence classes of $\approx^t$ are Mazurkiewicz traces.

We reduce preemption-safety under the abstract semantics to language inclusion modulo an independence relation. The independence relation I we use is defined on the set of abstract observable symbols as follows: $((tid, a, v, l), (tid', a', v', l')) \in I$ iff $tid \neq tid'$, and one of the following holds: (a) $v \neq v'$ or (b) $a \neq \mathsf{write} \wedge a' \neq \mathsf{write}$.

Proposition 1. *Given concurrent programs $\mathcal{C}$ and $\mathcal{C}'$, $[\![\mathcal{C}']\!]^P_{abs} \subseteq [\![\mathcal{C}]\!]^{NP}_{abs}$ iff $\mathcal{L}(\mathcal{A}([\![\mathcal{C}']\!]^P_{abs})) \subseteq \mathrm{Clo}_I(\mathcal{L}(\mathcal{A}([\![\mathcal{C}]\!]^{NP}_{abs})))$.*

4 Checking Language Inclusion

We first focus on the problem of language inclusion modulo an independence relation (Definition 1). This question corresponds to preemption-safety (Thm. 1, Prop. 1) and its solution drives our synchronization synthesis (Sec. 5).

Theorem 2. *For NFAs A, B over alphabet Σ and an independence relation $I \subseteq \Sigma \times \Sigma$, $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$ is undecidable [2].*

Fortunately, a bounded version of the problem is decidable. Recall the relation $\approx$ over Σ^* from Sec. 3.2. We define a symmetric binary relation $\approx_i$ over Σ^*: $(\sigma, \sigma') \in \approx_i$ iff $\exists (\alpha, \beta) \in I$: $(\sigma, \sigma') \in \approx$, $\sigma[i] = \sigma'[i+1] = \alpha$ and $\sigma[i+1] = \sigma'[i] = \beta$. Thus $\approx^i$ consists of all words that can be optained from each other by commuting the symbols at positions i and $i+1$. We next define a symmetric binary relation $\asymp$ over Σ^*: $(\sigma, \sigma') \in \asymp$ iff $\exists \sigma_1, \ldots, \sigma_t$: $(\sigma, \sigma_1) \in \approx_{i_1}, \ldots, (\sigma_t, \sigma') \in \approx_{i_{t+1}}$ and $i_1 < \ldots < i_{t+1}$. The relation $\asymp$ intuitively consists of words obtained from each other by making a single forward pass commuting multiple pairs of adjacent symbols. Let $\asymp^k$ denote the k-composition of $\asymp$ with itself. Given a language $\mathcal{L}$ over Σ, we use $\mathrm{Clo}_{k,I}(\mathcal{L})$ to denote the set $\{\sigma \in \Sigma^* : \exists \sigma' \in \mathcal{L}$ with $(\sigma, \sigma') \in \asymp^k\}$. In other words, $\mathrm{Clo}_{k,I}(\mathcal{L})$ consists of all words which can be generated from $\mathcal{L}$ using a finite-state transducer that remembers at most k symbols of its input words in its states.

Definition 2 (Bounded language inclusion modulo an independence relation). *Given NFAs A, B over Σ, $I \subseteq \Sigma \times \Sigma$ and a constant $k > 0$, the k-bounded language inclusion problem modulo I is: $\mathcal{L}(A) \subseteq \mathrm{Clo}_{k,I}(\mathcal{L}(B))$?*

Theorem 3. *For NFAs A, B over Σ, $I \subseteq \Sigma \times \Sigma$ and a constant $k > 0$, $\mathcal{L}(A) \subseteq \mathrm{Clo}_{k,I}(\mathcal{L}(B))$ is decidable.*

We present an algorithm to check k-bounded language inclusion modulo I, based on the antichain algorithm for standard language inclusion [9].

Antichain algorithm for language inclusion. Given a partial order $(X, \sqsubseteq)$, an antichain over X is a set of elements of X that are incomparable w.r.t. $\sqsubseteq$. In order to check $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$ for NFAs $A = (Q_A, \Sigma, \Delta_A, Q_{\iota,A}, F_A)$ and $B = (Q_B, \Sigma, \Delta_B, Q_{\iota,B}, F_B)$, the antichain algorithm proceeds by exploring A and B in lockstep. While A is explored nondeterministically, B is determinized on the fly for exploration. The algorithm maintains an antichain, consisting of tuples of the form (s_A, S_B), where $s_A \in Q_A$ and $S_B \subseteq Q_B$. The ordering relation

$\sqsubseteq$ is given by $(s_A, S_B) \sqsubseteq (s'_A, S'_B)$ iff $s_A = s'_A$ and $S_B \subseteq S'_B$. The algorithm also maintains a *frontier* set of tuples *yet* to be explored.

Given state $s_A \in Q_A$ and a symbol $\alpha \in \Sigma$, let $succ_\alpha(s_A)$ denote $\{s'_A \in Q_A : (s_A, \alpha, s'_A) \in \Delta_A\}$. Given set of states $S_B \subseteq Q_B$, let $succ_\alpha(S_B)$ denote $\{s'_B \in Q_B : \exists s_B \in S_B : \ (s_B, \alpha, s'_B) \in \Delta_B\}$. Given tuple (s_A, S_B) in the frontier set, let $succ_\alpha(s_A, S_B)$ denote $\{(s'_A, S'_B) : s'_A \in succ_\alpha(s_A), S'_B = succ_\alpha(s_B)\}$.

In each step, the antichain algorithm explores A and B by computing α-successors of all tuples in its current frontier set for all possible symbols $\alpha \in \Sigma$. Whenever a tuple (s_A, S_B) is found with $s_A \in F_A$ and $S_B \cap F_B = \varnothing$, the algorithm reports a counterexample to language inclusion. Otherwise, the algorithm updates its frontier set and antichain to include the newly computed successors using the two rules enumerated below. Given a newly computed successor tuple p':

- Rule 1: if there exists a tuple p in the antichain with $p \sqsubseteq p'$, then p' is not added to the frontier set or antichain,
- Rule 2: else, if there exist tuples $p_1, \ldots, p_n$ in the antichain with $p' \sqsubseteq p_1, \ldots, p_n$, then $p_1, \ldots, p_n$ are removed from the antichain.

The algorithm terminates by either reporting a counterexample, or by declaring success when the frontier becomes empty.

Antichain algorithm for k-bounded language inclusion modulo I. This algorithm is essentially the same as the standard antichain algorithm, with the automaton B above replaced by an automaton $B_{k,I}$ accepting $\mathrm{Clo}_{k,I}(\mathcal{L}(\mathrm{B}))$. The set $Q_{B_{k,I}}$ of states of $B_{k,I}$ consists of triples (s_B, η_1, η_2), where $s_B \in Q_B$ and η_1, η_2 are k-length words over Σ. Intuitively, the words η_1 and η_2 store symbols that are expected to be matched later along a run. The set of initial states of $B_{k,I}$ is $\{(s_B, \varnothing, \varnothing) : s_B \in I_B\}$. The set of final states of $B_{k,I}$ is $\{(s_B, \varnothing, \varnothing) : s_B \in F_B\}$. The transition relation $\Delta_{B_{k,I}}$ is constructed by repeatedly applying the following rules, in order, for each state (s_B, η_1, η_2) and each symbol α. In what follows, $\eta[\setminus i]$ denotes the word obtained from η by removing its i^{th} symbol.

1. Pick *new* s'_B and $\beta \in \Sigma$ such that $(s_B, \beta, s'_B) \in \Delta_B$
2. (a) If $\forall i$: $\eta_1[i] \neq \alpha$ and α is independent of all symbols in η_1, $\eta'_2 := \eta_2 \cdot \alpha$ and $\eta'_1 := \eta_1$, (b) else, if $\exists i$: $\eta_1[i] = \alpha$ and α is independent of all symbols in η_1 prior to i, $\eta'_1 := \eta_1[\setminus i]$ and $\eta'_2 := \eta_2$ (c) else, go to 1
3. (a) If $\forall i$: $\eta'_2[i] \neq \beta$ and β is independent of all symbols in η'_2, $\eta'_1 := \eta'_1 \cdot \beta$, (b) else, if $\exists i$: $\eta'_2[i] = \beta$ and β is independent of all symbols in η'_2 prior to i, $\eta'_2 := \eta'_2[\setminus i]$ (c) else, go to 1
4. Add $((s_B, \eta_1, \eta_2), \alpha, (s'_B, \eta'_1, \eta'_2))$ to $\Delta_{B_{k,I}}$ and go to 1.

Example 1. In Fig. 5, we have an NFA B with $\mathcal{L}(\mathrm{B}) = \{\alpha\beta, \beta\}$, $I = \{(\alpha, \beta)\}$ and $k = 1$. The states of $B_{k,I}$ are triples (q, η_1, η_2), where $q \in Q_B$ and $\eta_1, \eta_2 \in \{\varnothing, \alpha, \beta\}$. We explain the derivation of a couple of transitions of $B_{k,I}$. The transition shown in bold from $(q_0, \varnothing, \varnothing)$ on symbol β is obtained by applying the following rules once: 1. Pick q_1 since $(q_0, \alpha, q_1) \in \Delta_B$. 2(a). $\eta'_2 := \beta$, $\eta'_1 := \varnothing$. 3(a). $\eta'_1 := \alpha$. 4. Add $((q_0, \varnothing, \varnothing), \beta, (q_1, \alpha, \beta))$ to $\Delta_{B_{k,I}}$. The transition shown in bold from (q_1, α, β) on symbol α is obtained as follows: 1. Pick q_2 since $(q_1, \beta, q_2) \in \Delta_B$. 2(b). $\eta'_1 := \varnothing$, $\eta'_2 := \beta$. 3(b). $\eta'_2 := \varnothing$. 4. Add $((q_1, \alpha, \beta), \beta, (q_2, \varnothing, \varnothing))$ to $\Delta_{B_{k,I}}$. It can be seen that $B_{k,I}$ accepts the language $\{\alpha\beta, \beta\alpha, \beta\} = \mathrm{Clo}_{k,I}(B)$.

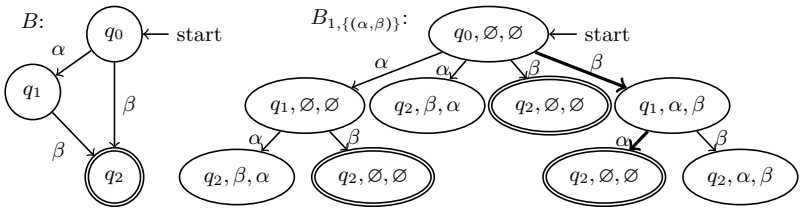

Fig. 5: Example for illustrating construction of $B_{k,I}$ for $k = 1$ and $I = \{(\alpha, \beta)\}$.

Proposition 2. *Given $k > 0$, NFA $B_{k,I}$ described above accepts* $\mathrm{Clo}_{k,I}(\mathcal{L}(B))$.

We develop a procedure to check language inclusion modulo I by iteratively increasing the bound k (see Appendix C for the complete algorithm). The procedure is *incremental*: the check for k+1-bounded language inclusion modulo I only explores paths along which the bound k was exceeded in the previous iteration.

5 Synchronization Synthesis

We now present our iterative synchronization synthesis procedure, which is based on the procedure in [11]. The reader is referred to [11] for further details. The synthesis procedure starts with the original program $\mathcal{C}$ and in each iteration generates a candidate synthesized program $\mathcal{C}'$. The candidate $\mathcal{C}'$ is checked for preemption-safety w.r.t. $\mathcal{C}$ under the abstract semantics, using our procedure for bounded language inclusion modulo I. If $\mathcal{C}'$ is found preemption-safe w.r.t. $\mathcal{C}$ under the abstract semantics, the synthesis procedure outputs $\mathcal{C}'$. Otherwise, an abstract counterexample *cex* is obtained. The counterexample is analyzed to infer additional synchronization to be added to $\mathcal{C}'$ for generating a new synthesized candidate.

The counterexample trace *cex* is a sequence of event identifiers: $tid_0.l_0; \ldots; tid_n.l_n$, where each l_i is a location identifier. We first analyze the *neighborhood* of *cex*, denoted *nhood*(*cex*), consisting of traces that are permutations of the events in *cex*. Note that each trace corresponds to an abstract observation sequence. Furthermore, note that preemption-safety requires the abstract observation sequence of any trace in *nhood*(*cex*) to be equivalent to that of some trace in *nhood*(*cex*) feasible under non-preemptive semantics. Let *bad traces* refer to the traces in *nhood*(*cex*) that are feasible under preemptive semantics and do not meet the preemption-safety requirement. The goal of our counterexample analysis is to characterize all bad traces in *nhood*(*cex*) in order to enable inference of synchronization fixes.

In order to succinctly represent subsets of *nhood*(*cex*), we use *ordering constraints*. Intuitively, ordering constraints are of the following forms: (a) atomic constraints $\Phi = A < B$ where A and B are events from *cex*. The constraint $A < B$ represents the set of traces in *nhood*(*cex*) where event A is scheduled before event B; (b) Boolean combinations of atomic constraints $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2$ and $\neg\Phi_1$. We have that $\Phi_1 \wedge \Phi_2$ and $\Phi_1 \vee \Phi_2$ respectively represent the intersection and union of the set of traces represented by Φ_1 and Φ_2, and that $\neg\Phi_1$ represents the complement (with respect to *nhood*(*cex*)) of the traces represented by Φ_1.

Non-preemptive neighborhood. First, we generate all traces in $nhood(cex)$ that are feasible under non-preemptive semantics. We represent a single trace π using an ordering constraint Φ_π that captures the ordering between non-independent accesses to variables in π. We represent all traces in $nhood(cex)$ that are feasible under non-preemptive semantics using the expression $\Phi = \bigvee_\pi \Phi_\pi$. The expression Φ acts as the correctness specification for traces in $nhood(cex)$.
Example. Recall the counterexample trace from the running example in Sec. 2: cex = $\texttt{T1.A}; \texttt{T2.A}; \texttt{T1.B}; \texttt{T1.C}; \texttt{T1.D}; \texttt{T2.B}; \texttt{T2.C}; \texttt{T2.D}$. There are two trace in $nhood(cex)$ that are feasible under non-preemptive semantics: π_1 = $\texttt{T1.A}; \texttt{T1.B}; \texttt{T1.C}; \texttt{T1.D}; \texttt{T2.A}; \texttt{T2.B}; \texttt{T2.C}; \texttt{T2.D}$ and π_2 = $\texttt{T2.A}; \texttt{T2.B}; \texttt{T2.C}; \texttt{T2.D}; \texttt{T1.A}; \texttt{T1.B}; \texttt{T1.C}; \texttt{T1.D}$. We represent π_1 as $\Phi(\pi_1) = \{\texttt{T1.A}, \texttt{T1.C}, \texttt{T1.D}\} < \texttt{T2.D} \wedge \texttt{T1.D} < \{\texttt{T2.A}, \texttt{T2.C}, \texttt{T2.D}\} \wedge \texttt{T1.B} < \texttt{T2.B}$ and π_2 as $\Phi(\pi_2) = \texttt{T2.D} < \{\texttt{T1.A}, \texttt{T1.C}, \texttt{T1.D}\} \wedge \{\texttt{T2.A}, \texttt{T2.C}, \texttt{T2.D}\} < \texttt{T1.D} \wedge \texttt{T2.B} < \texttt{T1.B}$. The correctness specification is $\Phi = \Phi(\pi_1) \vee \Phi(\pi_2)$.
Counterexample generalization. We next build a quantifier-free first order formula Ψ over the event identifiers in cex such that any model of Ψ corresponds to a bad trace in $nhood(cex)$. We iteratively enumerate models π of Ψ, building a constraint $\rho = \Phi(\pi)$ for each model π, and generalizing each ρ into ρ_g to represent a larger set of bad traces.
Example. Our trace cex from Sec. 2 would be generalized to $\texttt{T2.A} < \texttt{T1.D} \wedge \texttt{T1.D} < \texttt{T2.D}$. Any trace that fulfills this constraint is bad.
Inferring fixes. From each generalized formula ρ_g described above, we infer possible synchronization fixes to eliminate all bad traces satisfying ρ_g. The key observation we exploit is that common concurrency bugs often show up in our formulas as simple patterns of ordering constraints between events. For example, the pattern $tid_1.l_1 < tid_2.l_2 \wedge tid_2.l'_2 < tid_1.l'_1$ indicates an atomicity violation and can be rewritten into $\texttt{lock}(tid_1.[l_1 : l'_1], tid_2.[l_2 : l'_2])$. The complete list of such rewrite rules is in Appendix D. This list includes inference of locks and reordering of notify statements. The set of patterns we use for synchronization inference are not complete, i.e., there might be generalized formulae ρ_g that are not matched by any pattern. In practice, we found our current set of patterns to be adequate for most common concurrency bugs, including all bugs from the benchmarks in this paper. Our technique and tool can be easily extended with new patterns.
Example. The generalized constraint $\texttt{T2.A} < \texttt{T1.D} \wedge \texttt{T1.D} < \texttt{T2.D}$ matches the lock rule and yields $\texttt{lock}(\texttt{T2.[A : D]}, \texttt{T1.[D : D]})$. Since the lock involves events in the same function, the lock is merged into a single lock around instructions `A` and `D` in `open_dev_abs`. This lock is not sufficient to make the program preemption-safe. Another iteration of the synthesis procedure generates another counterexample for analysis and synchronization inference.

Proposition 3. *If our synthesis procedure generates a program $\mathcal{C}'$, then $\mathcal{C}'$ is preemption-safe with respect to $\mathcal{C}$.*

Note that our procedure does not guarantee that the synthesized program $\mathcal{C}'$ is deadlock-free. However, we avoid obvious deadlocks using heursitics such as merging overlapping locks. Further, our tool supports detection of any additional deadlocks introduced by synthesis, but relies on the user to fix them.

6 Implementation and Evaluation

We implemented our synthesis procedure in Liss. Liss is comprised of 5000 lines of C++ code and uses Clang/LLVM and Z3 as libraries. It is available as open-source software along with benchmarks at `https://github.com/thorstent/Liss`. The language inclusion algorithm is available separately as a library called Limi (`https://github.com/thorstent/Limi`). Liss implements the synthesis method presented in this paper with several optimizations. For example, we take advantage of the fact that language inclusion violations can often be detected by exploring only a small fraction of the input automata by constructing $\mathcal{A}([\![\mathcal{C}]\!]^{NP}_{abs})$ and $\mathcal{A}([\![\mathcal{C}]\!]^{P}_{abs})$ on the fly.

Our prototype implementation has several limitations. First, Liss uses function inlining and therefore cannot handle recursive programs. Second, we do not implement any form of alias analysis, which can lead to unsound abstractions. For example, we abstract statements of the form "`*x = 0`" as writes to variable `x`, while in reality other variables can be affected due to pointer aliasing. We sidestep this issue by manually massaging input programs to eliminate aliasing.

Finally, Liss implements a simplistic lock insertion strategy. Inference rules (see Sec. 5) produce locks expressed as sets of instructions that should be inside a lock. Placing the actual lock and unlock instructions in the C code is challenging because the instructions in the trace may span several basic blocks or even functions. We follow a structural approach where we find the innermost common parent block for the first and last instructions of the lock and place the lock and unlock instruction there. This does not work if the code has gotos or returns that could cause control to jump over the unlock statement. At the moment, we simply report such situations to the user.

We evaluate our synthesis method against the following criteria: (1) Effectiveness of synthesis from implicit specifications; (2) Efficiency of the proposed synthesis procedure; (3) Precision of the proposed coarse abstraction scheme on real-world programs.

Implicit vs explicit synthesis In order to evaluate the effectiveness of synthesis from implicit specifications, we apply Liss to the set of benchmarks used in our previous ConRepair tool for assertion-based synthesis [5]. In addition, we evaluate Liss and ConRepair on several *new* assertion-based benchmarks (Table 1). The set includes microbenchmarks modeling typical concurrency bug patterns in Linux drivers and the `usb-serial` macrobenchmark, which models a complete synchronization skeleton of the USB-to-serial adapter driver. We preprocess these benchmarks by eliminating assertions used as explicit specifications for synthesis. In addition, we replace statements of the form `assume(v)` with `await(v)`, redeclaring all variables `v` used in such statements as condition variables. This is necessary as our program syntax does not include `assume` statements.

We use Liss to synthesize a preemption-safe version of each benchmark. This method is based on the assumption that the benchmark is correct under non-preemptive scheduling and bugs can only arise due to preemptive scheduling. We discovered two benchmarks (`lc-rc.c` and `myri10ge.c`) that violated this assumption, i.e., they contained race conditions that manifested under non-

Name	LOC	Th	It	MB	BF(s)	Syn(s)	Ver(s)	CR(s)
ConRepair benchmarks [5]								
ex1.c	18	2	1	1	<1s	<1s	<1s	<1s
ex2.c	23	2	1	1	<1s	<1s	<1s	<1s
ex3.c	37	2	1	1	<1s	<1s	<1s	<1s
ex5.c	42	2	3	1	<1s	<1s	2s	<1s
lc-rc.c	35	4	0	1	-	-	<1s	9s
dv1394.c	37	2	1	1	<1s	<1s	<1s	17s
em28xx.c	20	2	1	1	<1s	<1s	<1s	<1s
f_acm.c	80	3	1	1	<1s	<1s	<1s	1871.99s
i915_irq.c	17	2	1	1	<1s	<1s	<1s	2.6s
ipath.c	23	2	1	1	<1s	<1s	<1s	12s
iwl3945.c	26	3	1	1	<1s	<1s	<1s	5s
md.c	35	2	1	1	<1s	<1s	<1s	1.5s
myri10ge.c	60	4	0	3	-	-	<1s	1.5s
usb-serial.bug1.c	357	7	2	1	0.4s	3.1s	3.4s	∞^b
usb-serial.bug2.c	355	7	1	3	0.7s	2.1s	12.9s	3563s
usb-serial.bug3.c	352	7	1	4	3.8s	1.3s	111.1s	∞^b
usb-serial.bug4.c	351	7	1	4	93.9s	2.4s	123.1s	∞^b
usb-serial.c[a]	357	7	0	4	-	-	103.2s	1200s
CPMAC driver benchmark								
cpmac.bug1.c	1275	5	1	1	1.3s	113.4s	21.9s	-
cpmac.bug2.c	1275	5	1	1	3.3s	68.4s	27.8s	-
cpmac.bug3.c	1270	5	1	1	5.4s	111.3s	8.7s	-
cpmac.bug4.c	1276	5	2	1	2.4s	124.8s	31.5s	-
cpmac.bug5.c	1275	5	1	1	2.8s	112.0s	58.0s	-
cpmac.c[a]	1276	5	0	1	-	-	17.4s	-

Th=Threads, It=Iterations, MB=Max bound, BF=Bug finding, Syn=Synthesis, Ver=Verification, Cr=ConRepair [a] bug-free example [b] timeout after 3 hours

Table 1: Experiments

preemptive scheduling; Liss did not detect these race conditions. Liss was able to detect and fix all other known races without relying on assertions. Furthermore, Liss detected a new race in the `usb-serial` family of benchmarks, which was not detected by ConRepair due to a missing assertion. We compared the output of Liss with manually placed synchronization (taken from real bug fixes) and found that the two versions were similar in most of our examples.

Performance and precision. ConRepair uses CBMC for verification and counterexample generation. Due to the coarse abstraction we use, both steps are much cheaper with Liss. For example, verification of `usb-serial.c`, which was the most complex in our set of benchmarks, took Liss 103 seconds, whereas it took ConRepair 20 minutes [5].

The loss of precision due to abstraction may cause the inclusion check to return a counterexample that is spurious in the concrete program, leading to unnecessary synchronization being synthesized. On our existing benchmarks, this only occurred once in the `usb-serial` driver, where abstracting away the return value of a function led to an infeasible trace. We refined the abstraction manually by introducing a condition variable to model the return value.

While this result is encouraging, synthetic benchmarks are not necessarily representative of real-world performance. We therefore implemented another set of benchmarks based on a complete Linux driver for the TI AR7 CPMAC Ethernet controller. The benchmark was constructed as follows. We manually preprocessed driver source code to eliminate pointer aliasing. We combined the driver with a model of the OS API and the software interface of the device written in C. We modeled most OS API functions as writes to a special memory location. Groups of unrelated functions were modeled using separate locations. Slightly more complex models were required for API functions that affect thread synchronization. For example, the `free_irq` function, which disables the driver's interrupt handler, blocks waiting for any outstanding interrupts to finish. Drivers can rely on this behavior to avoid races. We introduced a condition variable to model this synchronization. Similarly, most device accesses were modeled as writes to a special `ioval` variable. Thus, the only part of the device that required a more accurate model was its interrupt enabling logic, which affects the behavior of the driver's interrupt handler thread.

Our original model consisted of eight threads. Liss ran out of memory on this model, so we simplified it to five threads by eliminating parts of driver functionality. Nevertheless, we believe that the resulting model represents the most complex synchronization synthesis case study, based on real-world code, reported in the literature.

The CPMAC driver used in this case study did not contain any known concurrency bugs, so we artificially simulated five typical race conditions that commonly occur in drivers of this type [4]. Liss was able to detect and automatically fix each of these defects (bottom part of Table 1). We only encountered two program locations where manual abstraction refinement was necessary.

We conclude that (1) our coarse abstraction is highly precise in practice; (2) manual effort involved in synchronization synthesis can be further reduced via automatic abstraction refinement; (3) additional work is required to improve the performance of our method to be able to handle real-world systems without simplification. In particular, our analysis indicates that significant speed-up can be obtained by incorporating a partial order reduction scheme into the language inclusion algorithm.

7 Conclusion

We believe our approach and the encouraging experimental results open several directions for future research. Combining the abstraction refinement, verification (checking language inclusion modulo an independence relation), and synthesis (inserting synchronization) more tightly could bring improvements in efficiency. An additional direction we plan on exploring is automated handling of deadlocks, i.e., extending our technique to automatically synthesize deadlock-free programs. Finally, we plan to further develop our prototype tool and apply it to other domains of concurrent systems code.

References

1. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence - A static analysis approach to automatic fence insertion. In: CAV. pp. 508–524 (2014)
2. Bertoni, A., Mauri, G., Sabadini, N.: Equivalence and membership problems for regular trace languages. In: Automata, Languages and Programming, pp. 61–71. Springer (1982)
3. Bloem, R., Hofferek, G., Könighofer, B., Könighofer, R., Außerlechner, S., Spörk, R.: Synthesis of synchronization using uninterpreted functions. In: FMCAD. pp. 35–42 (2014)
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. Černý, P., Henzinger, T., Radhakrishna, A., Ryzhyk, L., Tarrach, T.: Regression-free synthesis for concurrency. In: CAV, pp. 568–584 (2014), `https://github.com/thorstent/ConRepair`
6. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI. pp. 304–315 (2008)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176 (2004), `http://www.cprover.org/cbmc/`
8. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. Springer (1982)
9. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: CAV. pp. 17–30. Springer (2006)
10. Deshmukh, J., Ramalingam, G., Ranganath, V., Vaswani, K.: Logical Concurrency Control from Sequential Proofs. In: Programming Languages and Systems, pp. 226–245 (2010)
11. Gupta, A., Henzinger, T., Radhakrishna, A., Samanta, R., Tarrach, T.: Succinct representation of concurrent trace sets. In: POPL15. pp. 433–444 (2015)
12. Jin, G., Zhang, W., Deng, D., Liblit, B., Lu, S.: Automated Concurrency-Bug Fixing. In: OSDI, pp. 221–236 (2012)
13. Ryzhyk, L., Chubb, P., Kuz, I., Heiser, G.: Dingo: Taming device drivers. In: Eurosys (Apr 2009)
14. Sadowski, C., Yi, J.: User evaluation of correctness conditions: A case study of cooperability. In: PLATEAU. pp. 2:1–2:6 (2010)
15. Solar-Lezama, A., Jones, C., Bodík, R.: Sketching concurrent data structures. In: PLDI. pp. 136–148 (2008)
16. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL. pp. 327–338 (2010)
17. Vechev, M.T., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: SAS. pp. 455–471 (2010)

A Semantics of preemptive and non-preemptive execution

In Fig. 6 we present the non-preemptive semantics. The preemptive semantics consist of the rules of the non-preemptive semantics and the single rule in Fig. 7.

We denote the state of a program as $\langle \mathcal{V}, ctid, (P_1, \ldots, P_n) \rangle$ where (a) Valuation $\mathcal{V}$ is a valuation of all program variables. Further, for each lock l, we have that $\mathcal{V}[l]$ holds the identifier of the thread that currently holds the lock, or 0 if no thread holds the lock. Similarly, for a condition variable c, we have that $\mathcal{V}[c] = 0$ if the variable is reset and $\mathcal{V}[c] = 1$ otherwise. (b) The value *ctid* is the thread identifier of the current executing thread or 0 in the initial state, and (c) Program fragments P_1 to P_n are the parts of the program to be executed by $\mathtt{T}_1$ to $\mathtt{T}_n$, respectively.

The premise in rule Sequential refers to the single-threaded semantics in Fig. 3 or the abstract single-threaded semantics in Fig. 4. Rules LockYield and AwaitYield force a context-switch iff the lock is not available or the condition variable is not set.

$$\frac{ctid = 0 \quad 1 \le ctid' \le n}{\langle \mathcal{V}, ctid, (P_1, \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (P_1, \ldots, P_n)\rangle}\textsc{ScheduleStart}$$

$$\frac{ctid = i \qquad \langle \mathcal{V}, P_i\rangle \xrightarrow{\alpha} \langle \mathcal{V}', P_i'\rangle}{\langle \mathcal{V}, ctid, (P_1, \ldots, P_i, \ldots, P_n)\rangle \xrightarrow{\alpha} \langle \mathcal{V}, ctid, (P_1, \ldots, P_i', \ldots, P_n)\rangle}\textsc{Sequential}$$

$$\frac{ctid = i \qquad \mathcal{V}(l) \notin \{0, i\} \qquad 1 \le ctid' \le n}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{lock}(l), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (P_1, \ldots, \mathsf{lock}(l), \ldots, P_n)\rangle}\textsc{LockYield}$$

$$\frac{ctid = i \qquad \mathcal{V}(l) \in \{0, i\}}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{lock}(l), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}[l := i], ctid, (P_1, \ldots, \mathsf{skip}, \ldots, P_n)\rangle}\textsc{Lock}$$

$$\frac{ctid = i \qquad \mathcal{V}(l) = ctid}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{unlock}(l), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}[l := 0], ctid, (P_1, \ldots, \mathsf{skip}, \ldots, P_n)\rangle}\textsc{Unlock}$$

$$\frac{ctid = i \qquad \mathcal{V}(c) = \mathsf{false} \qquad 1 \le ctid' \le n}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{await}(c), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{await}(c), \ldots, P_n)\rangle}\textsc{AwaitYield}$$

$$\frac{ctid = i \qquad \mathcal{V}(c) = \mathsf{true}}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{await}(c), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{skip}, \ldots, P_n)\rangle}\textsc{Await}$$

$$\frac{ctid = i}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{signal}(c), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}[c := \mathsf{true}], ctid, (P_1, \ldots, \mathsf{skip}, \ldots, P_n)\rangle}\textsc{Signal}$$

$$\frac{ctid = i}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{reset}(c), \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}[c := \mathsf{false}], ctid, (P_1, \ldots, \mathsf{skip}, \ldots, P_n)\rangle}\textsc{Reset}$$

$$\frac{s_1 \neq \mathsf{skip} \qquad \langle \mathcal{V}, ctid, (P_1, \ldots, s_1, \ldots, P_n)\rangle \xrightarrow{\alpha} \langle \mathcal{V}, ctid, (P_1, \ldots, s_1', \ldots, P_n)\rangle}{\langle \mathcal{V}, ctid, (P_1, \ldots, s_1; s_2, \ldots, P_n)\rangle \xrightarrow{\alpha} \langle \mathcal{V}, ctid, (P_1, \ldots, s_1'; s_2, \ldots, P_n)\rangle}\textsc{Sequence}$$

$$\frac{ctid = i \qquad 1 \le ctid' \le n \qquad P_i = \mathsf{skip}}{\langle \mathcal{V}, ctid, (P_1, \ldots, P_i, \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (P_1, \ldots, P_i, \ldots, P_n)\rangle}\textsc{DescheduleSkip}$$

$$\frac{ctid = i \qquad 1 \le ctid' \le n}{\langle \mathcal{V}, ctid, (P_1, \ldots, \mathsf{yield}, \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (P_1, \ldots, \mathsf{skip}, \ldots, P_n)\rangle}\textsc{Yield}$$

Fig. 6: Operational non-preemptive semantics

$$\frac{1 \leq ctid' \leq n}{\langle \mathcal{V}, ctid, (P_1, \ldots, P_n)\rangle \xrightarrow{\epsilon} \langle \mathcal{V}, ctid', (P_1, \ldots, P_n)\rangle}\textsc{DescheduePreempt}$$

Fig. 7: From non-preemptive semantics to preemptive semantics

B Proof of Thm. 1

Theorem 1. *Given concurrent program $\mathcal{C}$ and a synthesized program $\mathcal{C}'$ obtained by adding synchronization to $\mathcal{C}$, $[\![\mathcal{C}']\!]^P_{abs} \subseteq [\![\mathcal{C}]\!]^{NP}_{abs} \Rightarrow [\![\mathcal{C}']\!]^P \subseteq [\![\mathcal{C}]\!]^{NP}$.*

Proof. Let us assume $[\![\mathcal{C}']\!]^P_{abs} \subseteq [\![\mathcal{C}]\!]^{NP}_{abs}$.

Let σ' be a concrete observation sequence in $[\![\mathcal{C}']\!]^P$. Let σ'_{abs} be the abstract observation sequence in $[\![\mathcal{C}']\!]^P_{abs}$ corresponding to σ'. Then, there exists $\sigma_{abs} \in [\![\mathcal{C}]\!]^{NP}_{abs}$ such that σ_{abs} is equivalent to σ'_{abs}.

Observe that if two abstract observation sequences — σ'_{abs} from $[\![\mathcal{C}']\!]^P_{abs}$ and σ_{abs} from $[\![\mathcal{C}]\!]^{NP}_{abs}$ — are equivalent, then they correspond to executions over the *same* observable control-flow paths with the same data-flow into havoc and input/output statements. Hence, σ'_{abs} and σ_{abs} either both map back to infeasible concrete observation sequences, or both map back to feasible, *equivalent* concrete observation sequences.

Since σ'_{abs} maps back to a feasible concrete observation sequence σ' by definition, σ_{abs} also maps back to a feasible concrete observation sequence, say σ, such that σ is equivalent to σ'. Hence, we have $[\![\mathcal{C}']\!]^P \subseteq [\![\mathcal{C}]\!]^{NP}$. □

C Language Inclusion Procedure

The algorithm for k-bounded language inclusion modulo I is presented as function INCLUSION in Algo. 1 (ignore Lines 22-25 for now) . The function proceeds exactly as the standard antichain algorithm outlined earlier. It explores A nondeterministically as before, and $B_{k,I}$ is determinized on the fly for exploration. The antichain and frontier sets consist of tuples of the form $(s_A, S_{B_{k,I}})$, where $s_A \in Q_A$ and $S_{B_{k,I}} \subseteq Q_B \times \Sigma^k \times \Sigma^k$. Each tuple in the frontier set is first checked for equivalence w.r.t. acceptance (Line 18). If this check fails, the function reports language inclusion failure (Line 18). If this check succeeds, the successors are computed (Line 20). If a successor satisfies Rule 1, it is ignored (Line 21), otherwise it is added to the frontier (Line 26) and the antichain (Line 27). During the update of the antichain the algorithm ensures that its invariant is preserved according to rule 2. The frontier also stores a sequence of symbols that lead to a particular tuple of states in order to return a counterexample trace if language inclusion fails.

We develop a procedure to check language inclusion modulo I by iteratively increasing the bound k (see Algo. 1 in the appendix). The procedure is *incremental*: the check for $k+1$-bounded language inclusion modulo I only explores paths along which the bound k was exceeded in the previous iteration. Given a newly computed successor $(s'_A, S'_{B_{k,I}})$ for an iteration with bound k, if there exists some (s_B, η_1, η_2) in $S'_{B_{k,I}}$ such that the length of η_1 or η_2 exceeds k (Line 22), we remember the tuple $(s'_A, S'_{B_{k,I}})$ in the set *overflow* (Line 23). We continue exploration of $B_{k,I}$ from all states (s_B, η_1, η_2) with $|\eta_1| \leq k \wedge |\eta_2| \leq k$, but mark them *dirty*. If we find a counter-example to language inclusion we return it and test if it is spurious (Line 8). It may be a spurious counter-example caused because we removed states exceeding k. In that case we increase the bound to $k+1$, remove all dirty items from the antichain and frontier (lines 10-11), and add the items from the overflow (Line 12). Intuitively this will undo all exploration from the point(s) the bound was exceeded and restarts from that/those point(s).

To test if a particular counterexample is spurious, we invoke the language inclusion procedure, replacing the preemptive automaton with the exact trace (trace automaton) and allowing an infinite bound. This is fast and guaranteed to terminate as the trace automaton does not have loops. We found that this optimization helps find a valid counterexample faster.

Algorithm 1 Checking language inclusion modulo I

Require: Automata $A = (Q_A, \Sigma_A, \Delta_A, I_A, F_A)$ and $B = (Q_B, \Sigma_B, \Delta_B, I_B, F_B)$
Ensure: *true* only if $\mathcal{L}(A) \subseteq \mathrm{Clo}_I(\mathcal{L}(B))$, *false* only if $\mathcal{L}(A) \notin \mathrm{Clo}_I(\mathcal{L}(B))$

1: $\mathit{frontier} \leftarrow \{(s_A, \{(I_B, \varnothing, \varnothing)\}, \varnothing) : s_A \in I_A\}$
2: All tuples in *frontier* are not *dirty*
3: $\mathit{antichain} \leftarrow \mathit{frontier}$
4: $\mathit{overflow} \leftarrow \varnothing$
5: $k \leftarrow 2$
6: **while** *true* **do**
7: $\quad \mathit{cex} \leftarrow \textsc{inclusion}(k)$
8: $\quad$ **if** $\mathit{cex} \neq \mathit{true} \wedge \mathit{cex}$ is spurious **then**
9: $\quad\quad k \leftarrow k + 1$
10: $\quad\quad \mathit{frontier} \leftarrow \{(s_A, S_{B_{k,I}}) \in \mathit{frontier} : S_{B_{k,I}} \text{ not } \mathit{dirty}\} \cup \mathit{overflow}$
11: $\quad\quad \mathit{antichain} \leftarrow \{(s_A, S_{B_{k,I}}) \in \mathit{antichain} : S_{B_{k,I}} \text{ not } \mathit{dirty}\} \cup \mathit{overflow}$
12: $\quad\quad \mathit{overflow} \leftarrow \varnothing$
13: $\quad$ **else**
14: $\quad\quad$ **return** *cex*

15: **function** INCLUSION(k)
16: $\quad$ **while** $\mathit{frontier} \neq \varnothing$ **do**
17: $\quad\quad$ remove a tuple $(s_A, S_{B_{k,I}}, \mathit{cex})$ from *frontier*
18: $\quad\quad$ **if** $s_A \in F_A \wedge (S_{B_{k,I}} \cap F_B) = \varnothing$ **then return** *cex*
19: $\quad\quad$ **for all** $\alpha \in \Sigma$ **do**
20: $\quad\quad\quad (s'_A, S'_{B_{k,I}}) \leftarrow \mathit{succ}_\alpha(s_A, S_{B_{k,I}})$
21: $\quad\quad\quad$ **if** $\nexists p \in \mathit{antichain} : p \sqsubseteq (s'_A, S'_{B_{k,I}})$ **then** ▷ Rule 1
22: $\quad\quad\quad\quad$ **if** $\exists (s_B, \eta_1, \eta_2) \in S'_{B_{k,I}} : \ |\eta_1| > k \vee |\eta_2| > k$ **then**
23: $\quad\quad\quad\quad\quad$ **if** $S'_{B_{k,I}}$ not *dirty* **then** $\mathit{overflow} \leftarrow \mathit{overflow} \cup \{(s'_A, S'_{B_{k,I}})\}$
24: $\quad\quad\quad\quad\quad S'_{B_{k,I}} \leftarrow \{(s_B, \eta_1, \eta_2) \in S'_{B_{k,I}} : \ |\eta_1| \leq k \wedge |\eta_2| \leq k\}$
25: $\quad\quad\quad\quad\quad$ Mark $S'_{B_{k,I}}$ *dirty*
26: $\quad\quad\quad\quad \mathit{frontier} \leftarrow \mathit{frontier} \cup \{(s'_A, S'_{B_{k,I}}, \mathit{cex} \cdot \alpha)\}$
27: $\quad\quad\quad\quad \mathit{antichain} \leftarrow \mathit{antichain} \backslash \{p : S'_{B_{k,I}} \sqsubseteq p\} \cup \{(s'_A, S'_{B_{k,I}})\}$ ▷ Rule 2
28: $\quad$ **return** *true*

D Synchronization inference rules

The inference rules are applied as rewrite rules to the formula ρ_g obtained in Sec. 5. Each rule requires a certain subexpression in ρ_g and rewrites it to a synchronization primitive. That means that a single ρ_g could possibly be solved by one of several synchronization primitives.

The two lock rules fix atomicity violations and the reorder rule fixes ordering violations. The Add.Lock rule captures a set of threads where thread 1 is descheduled at or after location l_1 and thread 2 is scheduled at or before l_2. Another context switch deschedules thread 2 at or after l_2' and schedules again thread 1 at or before l_1'. As this pattern is present in the generalized ρ_g this context switch is necessary to make the trace bad. We can avoid this context switch by adding the lock from the conclusion. The Add.Lock2 rules captures the more general case where both, thread 2 interrupting thread 1 and thread 1 interrupting thread 2, are bad traces.

The Add.Reorder rule captures an ordering violation that can be fixed by moving a signal() statement. Intuitively the await() statement is signaled too early and thread 1 can start running in the preemptive semantics. In the non-preemptive semantics thread 2 keeps running after a signal() statement until a preemption point is reached.

$$\frac{\rho_g = tid_1.l_1 < tid_2.l_2' \ \wedge \ tid_2.l_2 < tid_1.l_1' \ \wedge \ \psi}{\texttt{lock}(tid_1.[l_1 : l_1'], tid_2.[l_2 : l_2']) \vee \psi} \ \textsc{Add.Lock}$$

$$\frac{\rho_g = tid_1.l_1 < tid_2.l_2 \ \wedge \ tid_2.l_2' < tid_1.l_1' \ \wedge \ \psi}{\texttt{lock}(tid_1.[l_1 : l_1'], tid_2.[l_2 : l_2']) \vee \psi} \ \textsc{Add.Lock2}$$

$$\frac{\begin{array}{lll} \rho_g = tid_1.l_1' < tid_2.l_2' \ \wedge \ \psi & \exists tid_1.l_1, tid_2.l_2 : & tid_1.l_1 < tid_1.l_1' \\ tid_2.l_2 < tid_2.l_2' & tid_1.l_1 = \mathsf{await}(c) & tid_2.l_2 = \mathsf{signal}(c) \end{array}}{\texttt{reorder}(tid_2.l_2, tid_2.l_2') \vee \psi} \ \textsc{Add.Reorder}$$

Fig. 8: Synchronization inference rules