Form Methods Syst Des (2017) 50:97–139
DOI 10.1007/s10703-016-0256-5

CrossMark

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[5] · Roopsha Samanta[6] · Thorsten Tarrach[3]

Published online: 27 September 2016
© The Author(s) 2016. This article is published with open access at Springerlink.com

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

This work was published, in part, in Computer Aided Verification (CAV) 2015 [4].

✉ Thorsten Tarrach
ttarrach@ist.ac.at

Pavol Černý
pavol.cerny@colorado.edu

Edmund M. Clarke
emc@cs.cmu.edu

Thomas A. Henzinger
tah@ist.ac.at

Arjun Radhakrishna
arjunrad@cis.upenn.edu

Leonid Ryzhyk
l.ryzhyk@samsung.com

Roopsha Samanta
roopsha@cs.purdue.edu

1 University of Colorado Boulder, 425 UCB, Boulder, CO 80309, USA

2 Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA

3 IST Austria, Am Campus 1, 3400 Klosterneuburg, Austria

4 University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104, USA

5 Samsung Research America, 665 Clyde Avenue, Mountain View, CA 94043, USA

6 University of Purdue, 610 Purdue Mall, West Lafayette, IN 47907, USA

a preemptive scheduler should be included in the set of sequences produced under a non-preemptive scheduler. We guarantee that our synthesis does not introduce deadlocks and that the synchronization inserted is optimal w.r.t. a given objective function. The solution is based on a finitary abstraction, an algorithm for bounded language inclusion modulo an independence relation, and generation of a set of global constraints over synchronization placements. Each model of the global constraints set corresponds to a correctness-ensuring synchronization placement. The placement that is optimal w.r.t. the given objective function is chosen as the synchronization solution. 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. The implicit specification helped us find one concurrency bug previously missed when model-checking using an explicit, user-provided specification. We implemented objective functions for coarse-grained and fine-grained locking and observed that different synchronization placements are produced for our experiments, favoring a minimal number of synchronization operations or maximum concurrency, respectively.

Keywords Synthesis · Concurrency · NFA language inclusion · MaxSAT

1 Introduction

Programming for a concurrent shared-memory system, such as most common computing devices today, is notoriously difficult and error-prone. Program synthesis for concurrency aims to mitigate this complexity by synthesizing synchronization code automatically [5,6,9, 15]. 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 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 (also known as *cooperative scheduling* [26]) can simplify the development of concurrent software, including operating system (OS) kernels, network servers, database systems, etc. [21,22]. In the non-preemptive 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.

A user evaluation by Sadowski and Yi [22] demonstrated that this model makes it easier for programmers to reason about and identify defects in concurrent code. There exist alternative implicit correctness specifications for concurrent programs. For example, for functional programs one can specify the final output of the sequential execution as the correct output. The synthesizer must then generate a concurrent program that is guaranteed to produce the same output as the sequential version [3]. This approach does not allow any form of thread coordination, e.g., threads cannot be arranged in a producer–consumer fashion. In addition,

it is not applicable to reactive systems, such as device drivers, where threads are not required to terminate.

Another implicit specification technique is based on placing *atomic sections* in the source code of the program [14]. In the synthesized program the computation performed by an atomic section must appear atomic with respect to the rest of the program. Specifications based on atomic sections and specifications based on the non-preemptive scheduling model, used by our tool, can be easily expressed in terms of each other. For example, one can simulate atomic sections by placing yield statements before and after each atomic section, as well as around every instruction that does not belong to any atomic section.

We believe that, at least for systems code, specifications based on the non-preemptive scheduling model are easier to write and are less error-prone than atomic sections. Atomic sections are subject to syntactic constraints. Each section is marked by a pair of matching opening and closing statements, which in practice means that the section must start and end within the same program block. In contrast, a yield can be placed anywhere in the program.

Moreover, atomic sections restrict the use of thread synchronization primitives such as semaphores. An atomic section either executes in its entirety or not at all. In the former case, all wait conditions along the execution path through the atomic section must be simultaneously satisfied *before* the atomic section starts executing. In practice, to avoid deadlocks, one can only place a blocking instruction at the start of an atomic section. Combined with syntactic constraints discussed above, this restricts the use of thread coordination with atomic sections—a severe limitation for systems code where thread coordination is common. In contrast, synchronization primitives can be used freely under non-preemptive scheduling. Internally, they are modeled using yields: for instance, a semaphore acquisition instruction is modeled by a yield followed by an assume statement that proceeds when the semaphore becomes available.

Lastly, our specification defaults to the safe choice of assuming everything needs to be atomic unless a yield statement is placed by the programmer. In contrast, code that uses atomic sections can be preempted at any point unless protected by an explicit atomic section.

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 $\mathscr{C}$ and a synthesized program $\mathscr{C}'$ obtained by adding synchronization to $\mathscr{C}$, $\mathscr{C}'$ is *preemption-safe* w.r.t. $\mathscr{C}$ if for each execution of $\mathscr{C}'$ under a preemptive scheduler, there is an observationally equivalent non-preemptive execution of $\mathscr{C}$. 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. The abstract program is denoted as $\mathscr{C}_{abs}$.

Two abstract program executions are observationally equivalent if they are equal modulo the classical independence relation I on memory accesses. This means that every sequence ω of observable actions is equivalent to a set of sequences of observable actions that are derived from ω by repeatedly commuting independent actions. Independent actions are

accesses to different locations, and accesses to the same location 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 NP_{abs}, representing the abstraction of the original program $\mathscr{C}$ under non-preemptive scheduling, and P_{abs}, representing the abstraction of the synthesized program $\mathscr{C}'$ under preemptive scheduling. We show that preemption-safety of $\mathscr{C}'$ w.r.t. $\mathscr{C}$ is implied by preemption-safety of the abstract synthesized program $\mathscr{C}'_{abs}$ w.r.t. the abstract original program $\mathscr{C}_{abs}$, which, in turn, is implied by language inclusion modulo I of NFAs P_{abs} and NP_{abs}. While the problem of language inclusion modulo an independence relation is undecidable [2], we show that the antichain-based algorithm for standard language inclusion [11] can be adapted to decide a bounded version of language inclusion modulo an independence relation.

Our synthesis works in a counterexample-guided inductive synthesis (CEGIS) loop that accumulates a set of global constraints. The loop starts with a counterexample obtained from the language inclusion check. A counterexample is a sequence of locations in $\mathscr{C}_{abs}$ such that their execution produce an observation sequence that is valid under the preemptive semantics, but not under the non-preemptive semantics. From the counterexample we infer *mutual exclusion (mutex) constraints*, which when enforced in the language inclusion check avoid returning the same counterexample again. We accumulate the mutex constraints from all counterexamples iteratively generated by the language inclusion check. Once the language inclusion check succeeds, we construct a set of global constraints using the accumulated mutex constraints and constraints for enforcing deadlock-freedom. This approach is the key difference to our previous work [4], where a greedy approach is employed that immediately places a lock to eliminate a bug. The greedy approach may result in a suboptimal lock placement with unnecessarily overlapping or nested locks.

The global approach allows us to use an objective function f to find an optimal lock placement w.r.t. f once all mutex constraints have been identified. Examples of objective functions include minimizing the number of lock statements (leading to coarse-grained locking) and maximizing concurrency (leading to fine-grained locking). We encode such an objective function, together with the global constraints, into a weighted maximum satisfiability (MaxSAT) problem, which is then solved using an off-the-shelf solver.

Since the synthesized lock placement is guaranteed not to introduce deadlocks our solution follows good programming practices with respect to locks: no double locking, no double unlocking and no locks locked at the end of the execution.

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, as well as an in-memory key-value store server. First, Liss was able to detect and eliminate all but two known concurrency bugs in our examples; these included one bug that we previously missed when synthesizing from explicit specifications [6], 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 for systems code: across all our benchmarks, we only encountered three program locations where manual abstraction refinement was needed to avoid the generation of unnecessary synchronization. Fourth, our tool finds a deadlock-free lock placement for both a fine-grained and a coarse-grained objective function. 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. With the two objective functions we implemented, Liss produces an optimal lock placements w.r.t. the objective.

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 without introducing deadlocks. 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. Fourth, we synthesize an optimal lock placement w.r.t. an objective function. 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.

2 Related work

This work is an extension of our work that appeared in CAV 2015 [4]. We included a proof for Theorem 3 that shows that language inclusion is undecidable for our particular construction of automata and independence relation. Further, we introduced a set of global mutex constraints that replace the greedy approach of our previous work and enables optimal lock placement according to an objective function.

Synthesis of synchronization is an active research area [3,5,6,8,12,15,17,23,24]. 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 explicitly. Further, correctness in Bloem et al. [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. Further, Khoshnood et al. developed ConcBugAssist [18], similar to our earlier paper [15], that employs a greedy loop to fix assertion violations in concurrent programs.

Our previous work [5,6,15] develops the trace-based synthesis algorithm. The input is a program with assertions in the code, which represent an explicit correctness specification. The algorithm proceeds in a loop where in each iteration a faulty trace is obtained using an external model checker. A trace is faulty if it violates the specification. The trace is subsequently generalized to a partial order [5,6] or a formula over happens-before relations [15], both representing a set of faulty traces. A formula over happens-before relations is basically a disjunction of partial orders. In our earlier previous work [5,6] the partial order is used to synthesize atomic sections and inner-thread reorderings of independent statements. In our later work [15] the happens-before formula is used to obtain locks, wait-signal statements, and barriers. The quality of the synthesized code heavily depends on how well the generalization steps works. Intuitively the more faulty traces are removed in one synthesis step the more general the solution is and the closer it is to the solution a human would have implemented.

The drawback of assertions as a specification is that it is hard to determine if a given set of assertions represents a complete specification. The current work does not rely on an external model-checker or an explicit specification. Here we are solving language inclusion,

a computationally harder problem than reachability. However, due to our abstraction, our tool performs significantly better than tools from our previous work [5,6], which are based on a mature model checker (CBMC [10]). Our abstraction is reminiscent of previously used abstractions that track reads and writes to individual locations (e.g., [1,25]). 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. For the trace generalization and synthesis we use the technique from our previous work [15] to infer looks. Due to our choice of specification no other synchronization primitives are needed.

In Vechev et al. [24] 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 [17] can detect and fix concurrency bugs by identifying simple bug patterns in the code.

The concepts of linearizability and serializability are very similar to our implicit specification. Linearizability [16] describes the illusion that every method of an object takes effect instantaneously at some point between the method call and return. A set of transactions is serializable [13,20] if they produce the same result, whether scheduled in parallel or in sequential order.

There has been a body of work on using a non-preemptive (cooperative) scheduler as an implicit specification. The notion of cooperability was introduced by Yi and Flanagan [26]. They require the user to annotate the program with yield statements to indicate thread interference. Then their system verifies that the yield specification is complete meaning that every trace is cooperable. A preemptive trace is cooperable if it is equivalent to a trace under the cooperative scheduler.

3 Illustrative example

Figure 2 contains our running example, a part of a device driver. A driver interfaces the operating system with the hardware device (as illustrated in Fig. 1) and may be used by different threads of the operating system in parallel. An operating system thread wishing to use the device must first call the OPEN_DEV procedure and finally the CLOSE_DEV procedure to indicate it no longer needs the device. The driver keeps track of the number of threads that interact with the device. The first thread to call OPEN_DEV will cause the driver to power up

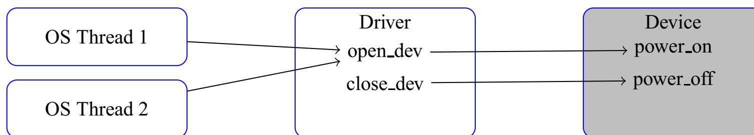

Fig. 1 Interaction of the device driver with the OS and the device

procedure OPEN_DEV
ℓ_1 **if** (open == 0) **then**
ℓ_2 POWER_UP
ℓ_3 open := open + 1
ℓ_4 yield

procedure CLOSE_DEV
ℓ_5 **if** (open > 0) **then**
ℓ_6 open := open − 1
ℓ_7 **if** (open == 0) **then**
ℓ_8 POWER_DOWN
ℓ_9 yield

Fig. 2 Running example

```
     procedure OPEN_DEV_ABS
ℓ1a    read(open)
ℓ1b    if (∗) then
ℓ2         write(dev)
ℓ3a    read(open)
ℓ3b    write(open)
ℓ4     yield
```

```
     procedure CLOSE_DEV_ABS
ℓ5a    read(open)
ℓ5b    if (∗) then
ℓ6a        read(open)
ℓ6b        write(open)
ℓ7a        read(open)
ℓ7b        if (∗) then
ℓ8             write(dev)
ℓ9     yield
```

Fig. 3 Abstraction of the running example

the device, the last thread to call CLOSE_DEV will cause the driver to power down the device. The interaction between the driver and the device are represented as procedure calls in lines ℓ_2 and ℓ_8. From the device's perspective, the power-on and power-off signals alternate. In general, we must assume that it is not safe to send the power-on signal twice in a row to the device. If executed with the non-preemptive scheduler the code in Fig. 2 will produce a sequence of a power-on signal followed by a power-off signal followed by a power-on signal and so on.

Consider the case where the procedure OPEN_DEV is called in parallel by two operating system threads that want to initiate usage of the device. Without additional synchronization, there could be two calls to POWER_UP in a row when executing under a preemptive scheduler. Consider two threads (T1 and T2) running the OPEN_DEV procedure. The corresponding trace is $\mathtt{T1}.\ell_1$; $\mathtt{T2}.\ell_1$; $\mathtt{T1}.\ell_2$; $\mathtt{T2}.\ell_2$; $\mathtt{T2}.\ell_3$; $\mathtt{T2}.\ell_4$; $\mathtt{T1}.\ell_3$; $\mathtt{T1}.\ell_4$. This sequence is not observationally equivalent to any sequence that can be produced when executing with a non-preemptive scheduler.

Figure 3 contains the abstracted versions of the two procedures, OPEN_DEV_ABS and CLOSE_DEV_ABS. For instance, the instruction $\mathtt{open} := \mathtt{open} + 1$ is abstracted to the two instructions labeled ℓ_{3a} and ℓ_{3b}. The calls to the device (POWER_UP and POWER_DOWN) are abstracted as writes to a hypothetical dev variable. This expresses the fact that interactions with the device are never independent. 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: $\mathtt{T1}.\ell_{1a}$; $\mathtt{T1}.\ell_{1b}$; $\mathtt{T2}.\ell_{1a}$; $\mathtt{T2}.\ell_{1b}$; $\mathtt{T1}.\ell_2$; $\mathtt{T2}.\ell_2$; $\mathtt{T2}.\ell_{3a}$; $\mathtt{T2}.\ell_{3b}$; $\mathtt{T2}.\ell_4$; $\mathtt{T1}.\ell_{3a}$; $\mathtt{T1}.\ell_{3b}$; $\mathtt{T1}.\ell_4$. Moreover, the trace cannot be transformed by swapping independent events into any trace possible under a non-preemptive scheduler. This is because instructions ℓ_{3b} : write(open) and ℓ_{1a} : read(open) are not independent. Further, ℓ_2 : write(dev) is not independent with itself. 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 stores it as a mutex constraint: mtx($[\ell_{1a}{:}\ell_{3b}]$, $[\ell_2{:}\ell_{3b}]$). Intuitively this constraint expresses the fact if one thread is executing any instruction between ℓ_{1a} and ℓ_{3b} no other thread may execute ℓ_2 or ℓ_{3b}.

While this constraint ensures two parallel calls to OPEN_DEV behave correctly, two parallel calls to CLOSE_DEV may result in the device receiving two POWER_DOWN signals. This is represented by the concrete trace $\mathtt{T1}.\ell_5$; $\mathtt{T1}.\ell_6$; $\mathtt{T2}.\ell_5$; $\mathtt{T2}.\ell_6$; $\mathtt{T2}.\ell_7$; $\mathtt{T2}.\ell_8$; $\mathtt{T2}.\ell_9$; $\mathtt{T1}.\ell_7$; $\mathtt{T1}.\ell_8$; $\mathtt{T1}.\ell_9$. The corresponding abstract trace is $\mathtt{T1}.\ell_{5a}$; $\mathtt{T1}.\ell_{5b}$; $\mathtt{T1}.\ell_{6a}$; $\mathtt{T1}.\ell_{6b}$; $\mathtt{T2}.\ell_{5a}$; $\mathtt{T2}.\ell_{5b}$; $\mathtt{T2}.\ell_{6a}$; $\mathtt{T2}.\ell_{6b}$; $\mathtt{T2}.\ell_{7a}$; $\mathtt{T2}.\ell_{7b}$; $\mathtt{T2}.\ell_8$; $\mathtt{T2}.\ell_9$; $\mathtt{T1}.\ell_{7a}$; $\mathtt{T1}.\ell_{7b}$; $\mathtt{T1}.\ell_8$; $\mathtt{T1}.\ell_9$. This trace is not possible under a non-preemptive scheduler and cannot be transformed to a trace possible under a non-preemptive scheduler. This results in a second mutex constraint mtx($[\ell_{5a}{:}\ell_8]$, $[\ell_{6b}{:}\ell_8]$). With both mutex constraints the program is correct. Our lock place-

```
    procedure OPEN_DEV
        lock(LkVar)
ℓ1      if (open == 0) then
ℓ2          POWER_UP
ℓ3      open := open + 1
        unlock(LkVar)
ℓ4      yield

    procedure CLOSE_DEV
        lock(LkVar)
ℓ5      if (open > 0) then
ℓ6          open := open − 1
ℓ7          if (open == 0) then
ℓ8              POWER_DOWN
        unlock(LkVar)
ℓ9      yield
```

Fig. 4 Running example with the synthesized locks

ment procedure then encodes these constraints in SMT and the models of the SMT formula are all the correct lock placements. In Fig. 4 we show OPEN_DEV and CLOSE_DEV with the inserted locks.

4 Formal framework and problem statement

We present the syntax and semantics of a *concrete* concurrent while language $\mathscr{W}$. For our solution strategy to be efficient we require an abstraction and we also introduce the syntax and semantics of the *abstract* concurrent while language $\mathscr{W}_{abs}$. While $\mathscr{W}$ (and our tool) permits non-recursive function call and return statements, we skip these constructs in the formalization below. We conclude the section by formalizing our notion of correctness for concrete concurrent programs.

4.1 Concrete concurrent programs

In our work, we assume a read or a write to a single shared variable executes atomically and further assume a sequentially consistent memory model.

4.1.1 Syntax of $\mathscr{W}$ (Fig. 5)

A concurrent program is a finite collection of threads $\langle \mathtt{T1}, \ldots, \mathtt{T}n \rangle$ where each thread is a statement written in the syntax of $\mathscr{W}$. Variables in $\mathscr{W}$ can be categorized into

- shared variables $ShVar_i$,
- thread-local variables $LoVar_i$,
- lock variables $LkVar_i$,
- condition variables $CondVar_i$ for wait-signal statements, and
- guard variables $GrdVar_i$ for assumptions.

The $LkVar_i$, $CondVar_i$ and $GrdVar_i$ variables are also shared between all threads. All variables range over integers with the exception of guard variables that range over Booleans (`true`, `false`). Each statement is labeled with a unique location identifier ℓ; we denote by $\mathsf{stmt}(\ell)$ the statement labeled by ℓ.

The language $\mathscr{W}$ includes standard sequential constructs, such as assignments, loops, conditionals, and goto statements. Additional statements control the interaction between threads, such as lock, wait-notify, and yield statements. In $\mathscr{W}$, we only permit expressions that read from at most one shared variable and assignments that either read from or write to

exactly one shared variable.[1] The language also includes assume, assume_not statements that operate on guard variables and become relevant later for our abstraction. The yield statement is in a sense an annotation as it has no effect on the actual program running under a preemptive scheduler. We still present it here because it has a semantic meaning under the non-preemptive scheduler.

Language $\mathscr{W}$ has two statements that allow communication with an external system: input(*ch*) reads from and output(*ch*, *ShExp*) writes to a communication channel *ch*. The channel is an interface between the program and an external system. The external system cannot observe the internal state of the program and only observes the information flow on the channel. In practice, we use the channels to model device registers. A device register is a special memory address, reading and writing from and to it is visible to the device. This is used to exchange information with a device. In our presentation, we assume all channels communicate with the same external system.

4.1.2 Semantics of $\mathscr{W}$

We first define the semantics of a single thread in $\mathscr{W}$, and then extend the definition to concurrent non-preemptive and preemptive semantics.

4.1.2.1 Single-thread semantics (Fig. 6) Let us fix a thread identifier *tid*. We use *tid* interchangeably with the program it represents. A state of a single thread is given by $\langle \mathscr{V}, \ell \rangle$ where $\mathscr{V}$ is a valuation of all program variables, and ℓ is a location identifier, indicating the statement in *tid* to be executed next. A thread is guaranteed not to read or write thread-local variables of other threads.

We define the *flow graph* $\mathscr{G}_{tid}$ for thread *tid* in a manner similar to the control-flow graph of *tid*. Every node of $\mathscr{G}_{tid}$ represents a single statement (basic blocks are not merged) and the node is labeled with the location ℓ of the statement. The flow graph $\mathscr{G}_{tid}$ has a unique entry node and a unique exit node. These two may coincide if the thread has no statements. The entry node is the first labeled statement in *tid*; we denote its location identifier by first_{tid}. The exit node is a special node corresponding to a hypothetical statement last_{tid} : skip placed at the end of *tid*.

We define successors of locations of *tid* using $\mathscr{G}_{tid}$. The location last has no successors. We define $\mathsf{succ}(\ell) = \ell'$ if node ℓ : *stmt* in $\mathscr{G}_{tid}$ has exactly one outgoing edge to node ℓ' : *stmt'*. Nodes representing conditionals and loops have two outgoing edges. We define $\mathsf{succ}_1(\ell) = \ell_1$ and $\mathsf{succ}_2(\ell) = \ell_2$ if node ℓ : *stmt* in $\mathscr{G}_{tid}$ has exactly two outgoing edges to nodes ℓ_1 : $stmt_1$ and ℓ_2 : $stmt_2$. Here succ_1 represents the then or the loop branch, whereas succ_2 represents the else or the loopexit branch.

We can now define the single-thread operational semantics. A single execution step $\langle \mathscr{V}, \ell \rangle \xrightarrow{\alpha} \langle \mathscr{V}', \ell' \rangle$ changes the program state from $\langle \mathscr{V}, \ell \rangle$ to $\langle \mathscr{V}', \ell' \rangle$, while optionally outputting an *observable symbol* α. The absence of a symbol is denoted using ϵ. In the following, e represents an expression and $e[v/\mathscr{V}[v]]$ evaluates an expression by replacing all variables v with their values in $\mathscr{V}$. We use $\mathscr{V}[v := k]$ to denote that variable v is set to k and all other variables in $\mathscr{V}$ remain unchanged.

[1] An expression/assignment statement that involves reading from/writing to multiple shared variables can always be rewritten into a sequence of atomic read/atomic write statements using local variables. For example the statement $\mathtt{x} := \mathtt{x} + 1$, where x is a global variable can be translated to $\mathtt{l} = \mathtt{x};\ \mathtt{x} = \mathtt{l} + 1$, where l is a fresh local variable.

$LbStmt ::=$	**Labeled Statement**
$\ell : stmt$	Statement annotated with a location
$LbStmt_1; LbStmt_2$	Sequence of statements
$stmt ::=$	Statement
skip	marks the end of the thread
$ShVar := LoExp$	Assignment to shared variable
$LoVar := ShExp$	Assignment to local variable
$ShVar :=$ havoc	Assign non-deterministic value
$ShVar := \mathsf{input}(ch)$	Read a value from channel *ch*
$\mathsf{output}(ch, ShExp)$	Write value of *ShExp* to channel *ch*
if $(ShExp)$ then $LbStmt_1$ else $LbStmt_2$	conditional
while $(ShExp)$ $LbStmt$	while loop
$\mathsf{lock}(LkVar)$	Locks the mutex lock
$\mathsf{unlock}(LkVar)$	Unlocks the mutex lock
$\mathsf{wait}(CondVar)$	Waits for *CondVar* to be signaled
$\mathsf{wait_not}(CondVar)$	Waits for *CondVar* to be reset
$\mathsf{signal}(CondVar)$	Notifies condition variable
$\mathsf{reset}(CondVar)$	Resets condition variable
$\mathsf{wait_reset}(CondVar)$	Waits and resets in an atomic operation
$\mathsf{assume}(GrdVar)$	Assume guard to be `true`
$\mathsf{assume_not}(GrdVar)$	Assume guard to be `false`
$GrdVar \leftarrow GrdExpr$	Assigns *GrdVar* the result of *GrdExpr*
yield	Allow current thread to be descheduled
$\mathsf{goto}(\ell)$	Set the instruction pointer to ℓ
$LoExp ::=$	Local-variable expression
c	Integer constant
$LoVar$	Thread-local variable
$op(LoExp_1, \ldots, LoExp_n)$	Operator application
$ShExp ::=$	Shared-variable expression
$LoExp$	Local-variable expression
$ShVar$	Shared variable
$op(ShVar, LoExp_1, \ldots, LoExp_n)$	Operator application with shared variable
$GrdExpr ::=$	Expression over guard variables
`true/false`	Boolean constant
$GrdVar$	Guard variable
$boolop(GrdExpr_1, \ldots, GrdExpr_n)$	Boolean operation

Fig. 5 Syntax of $\mathscr{W}$

In Fig. 6, we present the rules for single execution steps. Each step is atomic, no interference can occur while the expressions in the premise are being evaluated. The only rules with an observable output are:

1. HAVOC: Statement $\ell : ShVar :=$ havoc assigns shared variable *ShVar* a non-deterministic value (say k) and outputs the observable $(tid, \text{havoc}, k, ShVar)$.

2. INPUT, OUTPUT: ℓ : *ShVar* := input(*ch*) and ℓ : output(*ch*, *ShExp*) read and write values to the channel *ch*, and output (*tid*, in, k, *ch*) and (*tid*, out, k, *ch*), 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. The semantics of the synchronization statements shown in Fig. 6 is standard. The lock and unlock statements do not count and do not allow double (un)locking. There are no rules for goto and the sequence statement because they are already taken care of by the flow graph.

4.1.3 Concurrent semantics

A state of a concurrent program is given by $\langle \mathscr{V}, ctid, (\ell_1, \ldots, \ell_n)\rangle$ where $\mathscr{V}$ is a valuation of all program variables, *ctid* is the thread identifier of the currently executing thread and $\ell_1, \ldots, \ell_n$ are the locations of the statements to be executed next in threads $\texttt{T}_1$ to $\texttt{T}_n$, respectively. There are two additional states: $\langle\texttt{terminated}\rangle$ indicates the program has finished and $\langle\texttt{failed}\rangle$ indicates an assumption failed. Initially, all integer program variables and *ctid* equal 0, all guard variable equal `false` and for each $i \in [1, n] : \ell_i = \texttt{first}_i$. We introduce a non-preemptive and a preemptive semantics. The former is used as a specification of allowed executions, whereas the latter models concurrent sequentially consistent executions of the program.

4.1.3.1 Non-preemptive semantics (Fig. 7) The non-preemptive semantics ensures that a single thread from the program keeps executing using the single-thread semantics (Rule SEQ) until one of the following occurs: (a) the thread finishes execution (Rule THREAD_END) or (b) it encounters a yield, lock, wait or wait_not statement (Rule NSWITCH). In these cases, a context-switch is possible, however, the new thread must not be blocked. We consider a thread blocked if its current instruction is to acquire an unavailable lock, waits for a condition that is not signaled, or the thread reached the last location. Note the difference between wait/wait_not and assume/assume_not. The former allow for a context-switch while the latter transitions to the $\langle\texttt{failed}\rangle$ state if the assume is not fulfilled (rule ASSUME/ASSUME_NOT). A special rule exists for termination (Rule TERMINATE), which requires that all threads finished execution and also all locks are unlocked.

4.1.3.2 Preemptive semantics (Figs. 7, 8) 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. In particular, the preemptive semantics consist of the rules of the non-preemptive semantics and the single rule PSWITCH in Fig. 8.

4.2 Abstract concurrent programs

The state of the concrete semantics contains unbounded integer variables, which may result in an infinite state space. We therefore introduce a simple, data-oblivious abstraction $\mathscr{W}_{abs}$ for concurrent programs written in $\mathscr{W}$ communicating with an external system. The abstraction tracks types of accesses (read or write) to each memory location while abstracting away their values. Inputs/outputs to a channel 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 in the component on the other side of the interface. Havocs become ordinary writes to the variable they are assigned to. Every branch is taken non-deterministically and tracked. Given $\mathscr{C}$ written in $\mathscr{W}$, we denote by $\mathscr{C}_{abs}$ the corresponding abstract program written in $\mathscr{W}_{abs}$.

$$\frac{\mathsf{stmt}(\ell) = ShVar := LoExp \qquad LoExp[v/\mathscr{V}[v]] = k}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}[ShVar := k], \mathsf{succ}(\ell) \rangle} \text{ ASSIGNMENT}$$

$$\frac{\mathsf{stmt}(\ell) = ShVar := \mathsf{havoc} \qquad k \in \mathbb{Z}}{\langle \mathscr{V}, \ell \rangle \xrightarrow{(tid, \mathsf{havoc}, k, ShVar)} \langle \mathscr{V}[ShVar := k], \mathsf{succ}(\ell) \rangle} \text{ HAVOC}$$

$$\frac{\mathsf{stmt}(\ell) = ShVar := \mathsf{input}(ch) \qquad k \in \mathbb{Z}}{\langle \mathscr{V}, \ell \rangle \xrightarrow{(tid, \mathsf{in}, k, ch)} \langle \mathscr{V}[ShVar := k], \mathsf{succ}(\ell) \rangle} \text{ INPUT}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{output}(ch, ShExp) \quad ShExp[v/\mathscr{V}[v]] = k}{\langle \mathscr{V}, \ell \rangle \xrightarrow{(tid, \mathsf{out}, k, ch)} \langle \mathscr{V}, \mathsf{succ}(\ell) \rangle} \text{ OUTPUT}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\ (ShExp)\ \mathsf{then}\ LbStmt_1\ \mathsf{else}\ LbStmt_2 \qquad ShExp[v/\mathscr{V}[v]] \neq 0}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, \mathsf{succ}_1(\ell) \rangle} \text{ IF1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{if}\ (ShExp)\ \mathsf{then}\ LbStmt_1\ \mathsf{else}\ LbStmt_2 \qquad ShExp[v/\mathscr{V}[v]] = 0}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, \mathsf{succ}_2(\ell) \rangle} \text{ IF2}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{while}\ (ShExp)\ LbStmt \quad ShExp[v/\mathscr{V}[v]] \neq 0}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, \mathsf{succ}_1(\ell) \rangle} \text{ WHILE1}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{while}\ (ShExp)\ LbStmt \quad ShExp[v/\mathscr{V}[v]] = 0}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, \mathsf{succ}_2(\ell) \rangle} \text{ WHILE2}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{lock}(LkVar) \qquad \mathscr{V}[LkVar] = 0}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}[LkVar := tid], \mathsf{succ}(\ell) \rangle} \text{ LOCK}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{unlock}(LkVar) \qquad \mathscr{V}[LkVar] = tid}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}[LkVar := 0], \mathsf{succ}(\ell) \rangle} \text{ UNLOCK}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{wait}(CondVar)/\mathsf{wait_not}(CondVar) \quad \mathscr{V}[CondVar] = 1/0}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, \mathsf{succ}(\ell) \rangle} \text{ WAIT/WAIT_NOT}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{wait_reset}(CondVar) \qquad \mathscr{V}[CondVar] = 1}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}[CondVar := 0], \mathsf{succ}(\ell) \rangle} \text{ WAIT_RESET}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{signal}(CondVar)/\mathsf{reset}(CondVar)}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}[CondVar := 1/0], \mathsf{succ}(\ell) \rangle} \text{ SIGNAL/RESET}$$

$$\frac{\mathsf{stmt}(\ell) = \mathsf{assume}(GrdVar)/\mathsf{assume_not}(GrdVar) \quad \mathscr{V}[GrdVar] = \mathtt{true}/\mathtt{false}}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, \mathsf{succ}(\ell) \rangle} \text{ ASSUME/ASSUME_NOT}$$

$$\frac{\mathsf{stmt}(\ell) = GrdVar \leftarrow GrdExpr \quad GrdExpr[v/\mathscr{V}[v]] = k \quad k \in \{\mathtt{false}, \mathtt{true}\}}{\langle \mathscr{V}, \ell \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}[GrdVar := k], \mathsf{succ}(\ell) \rangle} \text{ SET GUARD}$$

Fig. 6 Single-thread semantics of $\mathscr{W}$

4.2.1 Abstract syntax (Fig. 9)

In the figure, *var* denotes all shared program variables and the `dev` variable. The syntax of all synchronization primitives and the assumptions over guard variables remains unchanged. The purpose of the guard variables is to improve the precision of our otherwise coarse abstraction.

$$\frac{ctid = i \qquad \langle \mathscr{V}, \ell_i \rangle \xrightarrow{\alpha} \langle \mathscr{V}', \ell_i' \rangle}{\langle \mathscr{V}, ctid, (\ldots, \ell_i, \ldots) \rangle \xrightarrow{\alpha} \langle \mathscr{V}', ctid, (\ldots, \ell_i', \ldots) \rangle} \text{ SEQ}$$

$$\frac{ctid = i \quad \ell_i = \mathsf{last}_i \quad ctid' \in \{1, \ldots, n\} \quad \neg\mathsf{blocked}(\ell_{ctid'}, \mathscr{V})}{\langle \mathscr{V}, ctid, (\ldots, \ell_i, \ldots) \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, ctid', (\ldots, \ell_i, \ldots) \rangle} \text{ THREAD_END}$$

$$\frac{\begin{array}{c}\mathsf{stmt}(\ell_i) = \mathsf{lock}(lk)/\mathsf{wait}(cv)/\mathsf{wait_not}(cv)/\mathsf{wait_reset}(cv)/\mathsf{yield} \\ ctid = i \qquad ctid' \in \{1, \ldots, n\} \qquad \neg\mathsf{blocked}(\ell_{ctid'}, \mathscr{V})\end{array}}{\langle \mathscr{V}, ctid, (\ldots, \ell_i, \ldots) \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, ctid', (\ldots, \ell_i, \ldots) \rangle} \text{ NSWITCH}$$

$$\frac{\forall i.\ \ell_i = \mathsf{last}_i \quad \forall j.\ \mathscr{V}[lk_j] = 0}{\langle \mathscr{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\varepsilon} \langle \mathtt{terminated} \rangle} \text{ TERMINATE}$$

$$\frac{\begin{array}{c}ctid = i \quad \mathsf{stmt}(\ell_i) = \mathsf{assume}(gv)/\mathsf{assume_not}(gv) \\ \mathscr{V}[gv] = 0/1\end{array}}{\langle \mathscr{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\varepsilon} \langle \mathtt{failed} \rangle} \text{ ASSUME/ASSUME_NOT}$$

$$\begin{aligned}\mathsf{blocked}(\ell, \mathscr{V}) = \; & (\mathsf{stmt}(\ell) = \mathsf{lock}(LkVar) \wedge \mathscr{V}[LkVar] \neq 0) \\ \vee \; & (\mathsf{stmt}(\ell) = \mathsf{wait}(CondVar) \wedge \mathscr{V}[CondVar] = 0) \\ \vee \; & (\mathsf{stmt}(\ell) = \mathsf{wait_not}(CondVar) \wedge \mathscr{V}[CondVar] = 1) \\ \vee \; & (\mathsf{stmt}(\ell) = \mathsf{wait_reset}(CondVar) \wedge \mathscr{V}[CondVar] = 0) \\ \vee \; & (\exists i :\ \ell = \mathsf{last}_i))\end{aligned}$$

Fig. 7 Non-preemptive semantics

$$\frac{ctid' \in \{1, \ldots, n\} \quad \neg\mathsf{blocked}(\ell_{ctid'}, \mathscr{V})}{\langle \mathscr{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle \xrightarrow{\varepsilon} \langle \mathscr{V}, ctid', (\ell_1, \ldots, \ell_n) \rangle} \text{ PSWITCH}$$

Fig. 8 Additional rule for preemptive semantics

Currently, they are inferred manually, but can presumably be inferred automatically using an iterative abstraction-refinement loop. In our current benchmarks, guard variables needed to be introduced in only three scenarios.

4.2.2 Abstraction function (Fig. 10)

A thread in $\mathscr{W}$ can be translated to $\mathscr{W}_{abs}$ using the abstraction function $\langle\!\langle\rangle\!\rangle$. The abstraction replaces all global variable access with $\mathsf{read}(var)$ and $\mathsf{write}(var)$ and replaces branching conditions with nondeterminism $(*)$. All synchronization primitives remain unaffected by the abstraction. The abstraction may result in duplicate labels ℓ, which are replaced by fresh labels. goto statements are reordered accordingly. Our abstraction records branching choices (branch tagging). If one were to remove branch-tagging, the abstraction would be unsound. The justification and intuition for this can be found further below in Theorem 1. For example in our running example in Fig. 2 the abstraction of ℓ_1 results in two abstract labels ℓ_{1a} and ℓ_{1b} in Fig. 3.

var ::=	Variables
$ShVar$	Shared variable
`dev`	Variable for interaction with channels
$LbStmt$ ::=	Labeled Statement
$\ell : stmt$	Statement annotated with a location
$LbStmt_1 ; LbStmt_2$	Sequence of statements
$stmt$ ::=	Statement
skip	marks the end of the thread
read(var)	Read a shared variable var
write(var)	Write to shared variable var
if $(*)$ then $LbStmt_1$ else $LbStmt_2$	conditional
while $(*)$ $LbStmt$	while loop
lock($LkVar$)	Locks the mutex lock
...	remaining statements as in Fig. 5

Fig. 9 Syntax of $\mathscr{W}_{abs}$

$$
\begin{aligned}
\langle\!\langle LoExp \rangle\!\rangle_\ell &= \textit{(nothing)} \\
\langle\!\langle ShVar \rangle\!\rangle_\ell &= \ell : \mathsf{read}(ShVar) \\
\langle\!\langle op(ShVar, LoExp_1, \ldots, LoExp_n) \rangle\!\rangle_\ell &= \ell : \mathsf{read}(ShVar) \\
\langle\!\langle LbStmt_1 ; LbStmt_2 \rangle\!\rangle &= \langle\!\langle LbStmt_1 \rangle\!\rangle ; \langle\!\langle LbStmt_2 \rangle\!\rangle \\
\langle\!\langle \ell : ShVar := LoExp \rangle\!\rangle &= \ell : \mathsf{write}(ShVar) \\
\langle\!\langle \ell : LoVar := ShExp \rangle\!\rangle &= \langle\!\langle ShExp \rangle\!\rangle_\ell \\
\langle\!\langle \ell : ShVar := \mathsf{havoc} \rangle\!\rangle &= \ell : \mathsf{write}(ShVar) \\
\langle\!\langle \ell : ShVar := \mathsf{input}(ch) \rangle\!\rangle &= \ell : \mathsf{write}(\mathtt{dev}) ; \ell : \mathsf{write}(ShVar) \\
\langle\!\langle \ell : \mathsf{output}(ShVar, ch) \rangle\!\rangle &= \ell : \mathsf{read}(ShVar) ; \ell : \mathsf{write}(\mathtt{dev}) \\
\langle\!\langle \ell : \mathsf{if}\ (ShExp)\ \mathsf{then}\ LbStmt_1\ \mathsf{else}\ LbStmt_2 \rangle\!\rangle &= \langle\!\langle ShExp \rangle\!\rangle_\ell ; \ell : \mathsf{if}\ (*)\ \mathsf{then}\ \langle\!\langle LbStmt_1 \rangle\!\rangle\ \mathsf{else}\ \langle\!\langle LbStmt_2 \rangle\!\rangle \\
\langle\!\langle \ell : \mathsf{while}\ (ShExp)\ LbStmt \rangle\!\rangle &= \langle\!\langle ShExp \rangle\!\rangle_\ell ; \ell : \mathsf{while}\ (*)\ \langle\!\langle LbStmt \rangle\!\rangle ; \langle\!\langle ShExp \rangle\!\rangle_\ell \\
\langle\!\langle \ell : \mathsf{lock}(LkVar) \rangle\!\rangle &= \ell : \mathsf{lock}(LkVar) \\
&\ldots
\end{aligned}
$$

Fig. 10 Abstraction function from $\mathscr{W}$ to $\mathscr{W}_{abs}$

4.2.3 Abstract semantics

As before, we first define the semantics of $\mathscr{W}_{abs}$ for a single-thread.

4.2.3.1 Single-thread semantics (Fig. 11) The abstract state of a single thread *tid* is given simply by $\langle \mathscr{V}_o, \ell \rangle$ where $\mathscr{V}_o$ is a valuation of all lock, condition and guard variables and ℓ is the location of the statement in *tid* to be executed next. We define the flow graph and successors for locations in the abstract program *tid* in the same way as before. An abstract observable symbol is of the form: (tid, θ, ℓ), where $\theta \in \{(\mathsf{read}, ShVar), (\mathsf{write}, ShVar), \mathsf{then}, \mathsf{else}, \mathsf{loop}, \mathsf{exitloop}\}$. The symbol θ records the type of access to variables along with the variable name $((\mathsf{read}, v), (\mathsf{write}, v))$ and records non-deterministic branching choices $\{\mathsf{if}, \mathsf{else}, \mathsf{loop}, \mathsf{exitloop}\}$. Fig. 11 presents the rules for statements unique to $\mathscr{W}_{abs}$; the rules for statements common to $\mathscr{W}_{abs}$ and $\mathscr{W}$ are the same.

4.2.3.2 Concurrent semantics A state of an abstract concurrent program is either $\langle \mathtt{terminated} \rangle$, $\langle \mathtt{failed} \rangle$, or is given by $\langle \mathscr{V}_o, ctid, (\ell_1, \ldots, \ell_n) \rangle$ where $\mathscr{V}_o$ is a valuation of all lock, condition and guard variables, *ctid* is the current thread identifier and $\ell_1, \ldots, \ell_n$ are the locations of the statements to be executed next in threads $\mathtt{T}_1$ to $\mathtt{T}_n$, respectively. The non-preemptive and preemptive semantics of a concurrent program written in $\mathscr{W}_{abs}$ are defined in the same way as that of a concurrent program written in $\mathscr{W}$.

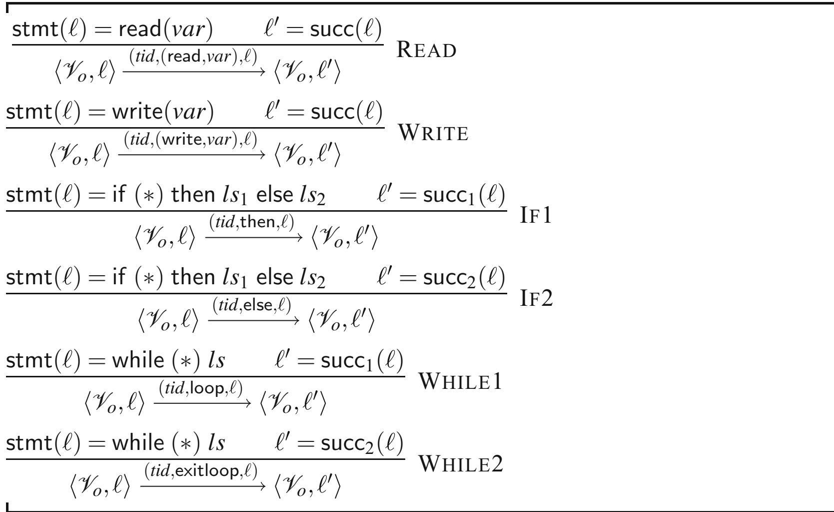

Fig. 11 Partial set of rules for single-thread semantics of $\mathscr{W}_{abs}$

4.3 Program correctness and problem statement

Let $\mathbb{W}$, $\mathbb{W}_{abs}$ denote the set of all concurrent programs in $\mathscr{W}$, $\mathscr{W}_{abs}$, respectively.

4.3.1 Executions

A *non-preemptive/preemptive execution* of a concurrent program $\mathscr{C}$ in $\mathbb{W}$ is an alternating sequence of program states and (possibly empty) observable symbols, $S_0\alpha_1 S_1 \ldots \alpha_k S_k$, such that (a) S_0 is the initial state of $\mathscr{C}$, (b) $\forall j \in [0, k-1]$, according to the non-preemptive/preemptive semantics of $\mathscr{W}$, we have $S_j \xrightarrow{\alpha_{j+1}} S_{j+1}$, and (c) S_k is the state $\langle \texttt{terminated} \rangle$. A non-preemptive/preemptive execution of a concurrent program $\mathscr{C}_{abs}$ in $\mathbb{W}_{abs}$ is defined in the same way, replacing the corresponding semantics of $\mathscr{W}$ with that of $\mathscr{W}_{abs}$.

4.3.2 Observable behaviors

Let π be an execution of program $\mathscr{C}$ in $\mathbb{W}$, then we denote with $\omega = \mathsf{obs}(\pi)$ the sequence of non-empty observable symbols in π. We use $[\![\mathscr{C}]\!]^{NP}$, resp. $[\![\mathscr{C}]\!]^{P}$, to denote the *non-preemptive*, resp. *preemptive*, *observable behavior* of $\mathscr{C}$, that is all sequences $\mathsf{obs}(\pi)$ of all executions π under the non-preemptive, resp. preemptive, scheduling. The *non-preemptive/preemptive observable behavior* of program $\mathscr{C}_{abs}$ in $\mathbb{W}_{abs}$, denoted $[\![\mathscr{C}_{abs}]\!]^{NP}/[\![\mathscr{C}_{abs}]\!]^{P}$, is defined similarly.

We specify correctness of concurrent programs in $\mathbb{W}$ using two *implicit* criteria, presented below.

4.3.3 Preemption-safety

Observable behaviors ω_1 and ω_2 of a program $\mathscr{C}$ in $\mathbb{W}$ are *equivalent* if: (a) the subsequences of ω_1 and ω_2 containing only symbols of the form (tid, in, k, t) and $(tid, \mathsf{out}, k, t)$ are equal

and (b) for each thread identifier *tid*, the subsequences of ω_1 and ω_2 containing only symbols of the form $(tid, \textsf{havoc}, k, x)$ are equal. Intuitively, observable behaviors are equivalent if they have the same interaction with the interface, and the same non-deterministic choices in each thread. For sets $\mathscr{O}_1$ and $\mathscr{O}_2$ of observable behaviors, we write $\mathscr{O}_1 \Subset \mathscr{O}_2$ to denote that each sequence in $\mathscr{O}_1$ has an equivalent sequence in $\mathscr{O}_2$.

Given concurrent programs $\mathscr{C}$ and $\mathscr{C}'$ in $\mathbb{W}$ such that $\mathscr{C}'$ is obtained by adding locks to $\mathscr{C}$, $\mathscr{C}'$ is *preemption-safe* w.r.t. $\mathscr{C}$ if $[\![\mathscr{C}']\!]^P \Subset [\![\mathscr{C}]\!]^{NP}$.

4.3.4 Deadlock-freedom

A state S of concurrent program $\mathscr{C}$ in $\mathbb{W}$ is a *deadlock state* under non-preemptive/preemptive semantics if

(a) The repeated application of the rules of the non-preemptive/preemptive semantics from the initial state S_0 of $\mathscr{C}$ can lead to S,
(b) $S \neq \langle \texttt{terminated} \rangle$,
(c) $S \neq \langle \texttt{failed} \rangle$, and
(d) $\neg \exists S' \colon \langle S \rangle \xrightarrow{\alpha} \langle S' \rangle$ according to the non-preemptive/preemptive semantics of $\mathscr{W}$.

Program $\mathscr{C}$ in $\mathbb{W}$ is *deadlock-free under non-preemptive/preemptive semantics* if no non-preemptive/preemptive execution of $\mathscr{C}$ hits a deadlock state. In other words, every non-preemptive/preemptive execution of $\mathscr{C}$ ends in state $\langle \texttt{terminated} \rangle$ or $\langle \texttt{failed} \rangle$. The $\langle \texttt{failed} \rangle$ state indicates an assumption did not hold, which we do not consider a deadlock. We say $\mathscr{C}$ is *deadlock-free* if it is deadlock-free under both non-preemptive and preemptive semantics.

4.3.5 Problem statement

We are now ready to state our main problem, the optimal synchronization synthesis problem. We assume we are given a cost function f from a program $\mathscr{C}'$ to the cost of the lock placement solution, formally $f : \mathbb{W} \mapsto \mathbb{R}$. Then, given a concurrent program $\mathscr{C}$ in $\mathbb{W}$, the goal is to synthesize a new concurrent program $\mathscr{C}'$ in $\mathbb{W}$ such that:

(a) $\mathscr{C}'$ is obtained by adding locks to $\mathscr{C}$,
(b) $\mathscr{C}'$ is preemption-safe w.r.t. $\mathscr{C}$,
(c) $\mathscr{C}'$ has no deadlocks not present in $\mathscr{C}$, *and*,
(d) $\mathscr{C}' = \underset{\mathscr{C}'' \in \mathbb{W} \text{ satisfying (a)-(c) above}}{\arg\min} f(\mathscr{C}'')$

5 Solution overview

Our solution framework (Fig. 12) consists of the following main components. We briefly describe each component below and then present them in more detail in subsequent sections.

5.1 Reduction of preemption-safety to language inclusion

To ensure tractability of checking preemption-safety, we build the abstract program $\mathscr{C}_{abs}$ from $\mathscr{C}$ using the abstraction function described in Sect. 4.2. Under abstraction, we model each thread as a nondeterministic finite automaton (NFA) over a finite alphabet consisting of

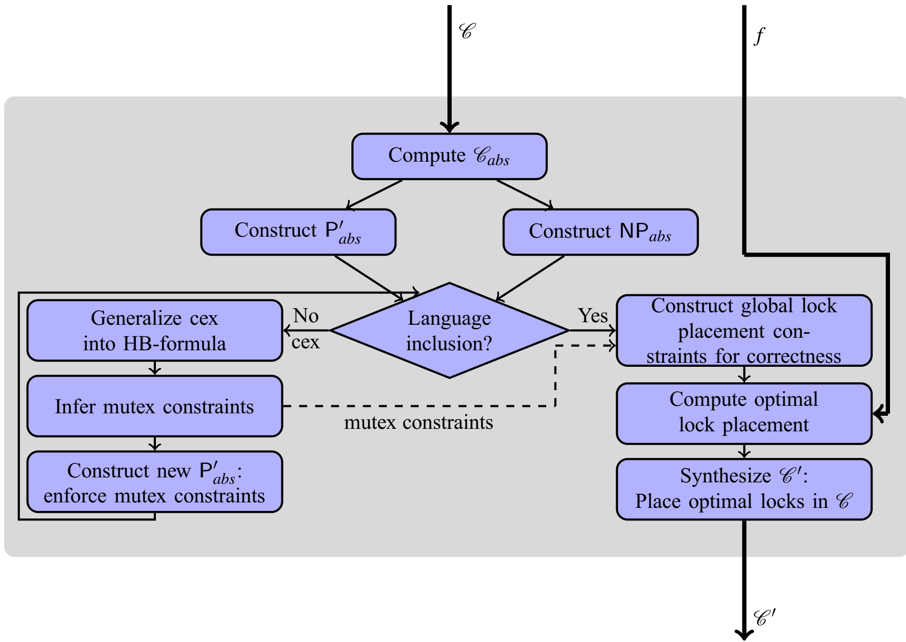

Fig. 12 Solution overview

abstract observable symbols. This enables us to construct NFAs NP_{abs} and P'_{abs} accepting the languages $[\![\mathscr{C}_{abs}]\!]^{NP}$ and $[\![\mathscr{C}'_{abs}]\!]^{P}$, respectively. We proceed to check if all words of P'_{abs} are included in NP_{abs} modulo an independence relation I that respects the equivalence of observables. We describe the reduction of preemption-safety to language inclusion and our language inclusion check procedure in Sect. 6.

5.2 Inference of mutex constraints from generalized counterexamples

If P'_{abs} and NP_{abs} do not satisfy language inclusion modulo I, then we obtain a counterexample *cex*. A counterexample is a sequence of locations an observation sequence that is in $[\![\mathscr{C}_{abs}]\!]^{P}$, but not in $[\![\mathscr{C}'_{abs}]\!]^{NP}$. We analyze *cex* to infer constraints on $\mathscr{L}(\mathsf{P}'_{abs})$ for eliminating *cex*. We use *nhood*(*cex*) to denote the set of all permutations of the symbols in *cex* that are accepted by P'_{abs}. Our counterexample analysis examines the set *nhood*(*cex*) to obtain an *hbformula* ϕ—a Boolean combination of *happens-before* ordering constraints between events—representing all counterexamples in *nhood*(*cex*). Thus *cex* is generalized into a larger set of counterexamples represented as ϕ. From ϕ, we infer possible *mutual exclusion (mutex) constraints* on $\mathscr{L}(\mathsf{P}'_{abs})$ that can eliminate all counterexamples satisfying ϕ. We describe the procedure for finding constraints from *cex* in Sect. 7.1.

5.3 Automaton modification for enforcing mutex constraints

Once we have the mutex constraints inferred from a generalized counterexample, we enforce them in P'_{abs}, effectively removing transitions from the automaton that violate the mutex constraint. This completes our loop and we repeat the language inclusion check of P'_{abs} and NP_{abs}. If another counterexample is found our loop continues, if the language inclusion check

succeeds we proceed to the lock placement. This differs from the greedy approach employed in our previous work [4] that modifies $\mathscr{C}'_{abs}$ and then constructs a new automaton P'_{abs} from $\mathscr{C}'_{abs}$ before restarting the language inclusion. The greedy approach inserts locks into $\mathscr{C}'_{abs}$ that are never removed in a future iteration. This can lead to inefficient lock placement. For example a larger lock may be placed that completely surrounds an earlier placed lock.

5.4 Computation of an f-optimal lock placement

Once P'_{abs} and NP_{abs} satisfy language inclusion modulo I, we formulate global constraints over lock placements for ensuring correctness. These global constraints include all mutex constraints inferred over all iterations and constraints for enforcing deadlock-freedom. Any model of the global constraints corresponds to a lock placement that ensures program correctness. We describe the formulation of these global constraints in Sect. 8.

Given a cost function f, we compute a lock placement that satisfies the global constraints and is optimal w.r.t. f. We then synthesize the final output $\mathscr{C}'$ by inserting the computed lock placement in $\mathscr{C}$. We present various objective functions and describe the computation of their respective optimal solutions in Sect. 9.

6 Checking preemption-safety

6.1 Reduction of preemption-safety to language inclusion

6.1.1 Soundness of the abstraction

Formally, two observable behaviors $\omega_1 = \alpha_0 \ldots \alpha_k$ and $\omega_2 = \beta_0 \ldots \beta_k$ of an abstract program $\mathscr{C}_{abs}$ in $\mathbb{W}_{abs}$ are *equivalent* if:

(A1) For each thread *tid*, the subsequences of $\alpha_0 \ldots \alpha_k$ and $\beta_0 \ldots \beta_k$ containing only symbols of the form (tid, a, ℓ), for all a, are equal,
(A2) For each variable var, the subsequences of $\alpha_0 \ldots \alpha_k$ and $\beta_0 \ldots \beta_k$ containing only write symbols (of the form $(tid, (\mathsf{write}, var), \ell)$) are equal, and
(A3) For each variable var, the multisets of symbols of the form $(tid, (\mathsf{read}, var), \ell)$ between any two write symbols, as well as before the first write symbol and after the last write symbol are identical.

Using this notion of equivalence, the notion of preemption-safety is extended to abstract programs: Given abstract concurrent programs $\mathscr{C}_{abs}$ and $\mathscr{C}'_{abs}$ in $\mathbb{W}_{abs}$ such that $\mathscr{C}'_{abs}$ is obtained by adding locks to $\mathscr{C}_{abs}$, $\mathscr{C}'_{abs}$ is *preemption-safe* w.r.t. $\mathscr{C}_{abs}$ if $[\![\mathscr{C}'_{abs}]\!]^P \Subset_{abs} [\![\mathscr{C}_{abs}]\!]^{NP}$.

For the abstraction to be sound we require only that whenever preemption-safety does not hold for a program $\mathscr{C}$, then there must be a trace in its abstraction $\mathscr{C}_{abs}$ feasible under preemptive, but not under non-preemptive semantics.

To illustrate this we use the program in Fig. 13, which is not preemption-safe. To see this consider the observation (`T1`, out, 10, `ch`) that cannot occur in the non-preemptive semantics because `x` is always 0 at ℓ_4. Note that ℓ_3 is unreachable because the variable `y` is initialized to 0 and never assigned. With the preemptive semantics the output can be observed if thread `T2` interrupts thread `T1` between lines ℓ_1 and ℓ_4. An example trace would be $\ell_1;\ \ell_6;\ \ell_2;\ \ell_4;\ \ell_5$.

If we consider the abstract semantics, we notice that under the non-preemptive abstract semantics ℓ_3 is reachable because the abstraction makes the branching condition in ℓ_2 non-deterministic. However, since our abstraction is sound there must still be an obser-

Fig. 13 Example showing how the abstraction works

```
x := 0; y := 0
Thread T1                         Thread T2
ℓ1  x := 0                        ℓ6  x := 1
ℓ2  if (y) then
ℓ3      yield
ℓ4  if (x) then
ℓ5      output(ch, 10)
```

vation sequence that is observable under the abstract preemptive semantics, but not under the abstract non-preemptive semantics. This observation sequence is $(\texttt{T1}, (\textsf{write}, \texttt{x}), \ell_1)$, $(\texttt{T2}, (\textsf{write}, \texttt{x}), \ell_6)$, $(\texttt{T1}, (\textsf{read}, \texttt{y}), \ell_2)$, $(\texttt{T1}, \textsf{else}, \ell_2)$, $(\texttt{T1}, (\textsf{read}, \texttt{x}), \ell_4)$, $(\texttt{T1}, \textsf{then}, \ell_2)$, $(\texttt{T1}, (\textsf{write}, \texttt{dev}), \ell_5)$. The branch tagging records that the else branch is taken in ℓ_2. The non-preemptive semantics cannot produce this observation sequences because it must also take the else branch in ℓ_2 and can therefore not reach the yield statement and context-switch. As a site note, it is also not possible to transform this observation sequence into an equivalent one under the non-preemptive semantics because of the write to x at ℓ_6 and the accesses to x in ℓ_1 and ℓ_4.

This example illustrates why branch tagging is crucial to soundness of the abstraction. If we assume a hypothetical abstract semantics without branch tagging we would get the following preemptive observation sequence: $(\texttt{T1}, (\textsf{write}, \texttt{x}), \ell_1)$, $(\texttt{T2}, (\textsf{write}, \texttt{x}), \ell_6)$, $(\texttt{T1}, (\textsf{read}, \texttt{y}), \ell_2)$, $(\texttt{T1}, (\textsf{read}, \texttt{x}), \ell_4)$, $(\texttt{T1}, (\textsf{write}, \texttt{dev}), \ell_5)$. This sequence would also be a valid observation sequence under the non-preemptive semantics, because it could take the then branch in ℓ_2 and reach the yield statement and context-switch.

Theorem 1 (soundness) *Given concurrent program $\mathscr{C}$ and a synthesized program $\mathscr{C}'$ obtained by adding locks to $\mathscr{C}$, $[\![\mathscr{C}'_{abs}]\!]^P \Subset_{abs} [\![\mathscr{C}_{abs}]\!]^{NP} \implies [\![\mathscr{C}']\!]^P \Subset [\![\mathscr{C}]\!]^{NP}$.*

Proof It is easier to prove the contrapositive: $[\![\mathscr{C}']\!]^P \not\Subset [\![\mathscr{C}]\!]^{NP} \implies [\![\mathscr{C}'_{abs}]\!]^P \not\Subset_{abs} [\![\mathscr{C}_{abs}]\!]^{NP}$.

$[\![\mathscr{C}']\!]^P \not\Subset [\![\mathscr{C}]\!]^{NP}$ means that there is an observation sequence ω' of $[\![\mathscr{C}']\!]^P$ with no equivalent observation sequence in $[\![\mathscr{C}]\!]^{NP}$. We now show that the abstract sequence ω'_{abs} in $[\![\mathscr{C}'_{abs}]\!]^P$ corresponding to the sequence ω' has no equivalent sequence in $[\![\mathscr{C}_{abs}]\!]^{NP}$.

Towards contradiction we assume there is such an equivalent sequence ω_{abs} in $[\![\mathscr{C}_{abs}]\!]^{NP}$. We show that if ω_{abs} indeed existed it would correspond to a concrete sequence ω that is equivalent to ω', thereby contradicting our assumption.

By (*A1*) ω_{abs} would have the same control flow as ω'_{abs} because of the branch tagging. By (*A2*) and (*A3*) ω_{abs} would have the same data-flow, meaning all reads from global variables are reading the values written by the same writes as in ω'_{abs}. Since all interactions with the environment are abstracted to $\textsf{write}(\texttt{dev})$ the order of interactions must be the same between ω_{abs} and ω'_{abs}. This means that, assuming all inputs and havocs are returning the same value, in the execution ω corresponding to ω_{abs} all variables valuation are identical to those in ω'. Therefore, ω is feasible and its interaction with the environment is identical to ω' as all variable valuations are identical. Identical interaction with the environment is how equivalence between ω and ω' is defined. This concludes our proof. □

6.1.2 Language inclusion modulo an independence relation

We define the problem of 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 $\approx_I$ over words in Σ^*: for all words $\sigma, \sigma' \in \Sigma^*$ and $(\alpha, \beta) \in I$, $(\sigma \cdot \alpha\beta \cdot \sigma', \sigma \cdot \beta\alpha \cdot \sigma') \in \approx_I$. Let $\approx_I^t$ denote the reflexive transitive closure of $\approx_I$.[2] Given a language $\mathscr{L}$ over Σ, the closure of $\mathscr{L}$ w.r.t. I, denoted $\mathrm{Clo}_I(\mathscr{L})$, is the set $\{\sigma \in \Sigma^* : \exists\sigma' \in \mathscr{L}$ with $(\sigma, \sigma') \in \approx_{\mathscr{I}}\}$. Thus, $\mathrm{Clo}_I(\mathscr{L})$ consists of all words that can be obtained from some word in $\mathscr{L}$ 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: $\mathscr{L}(\mathrm{A}) \subseteq \mathrm{Clo}_I(\mathscr{L}(\mathrm{B}))$?

6.1.3 Data independence relation

We define the data independence relation I_D over our observable symbols. Two symbols $\alpha = (tid_\alpha, a_\alpha, \ell_\alpha)$ and $\beta = (tid_\beta, a_\beta, \ell_\beta)$ are independent, $(\alpha, \beta) \in I_D$, iff (I0) $tid_\alpha \neq tid_\beta$ and one of the following hold:

(I1) a_α or a_β in $\{\mathsf{then}, \mathsf{else}, \mathsf{loop}, \mathsf{loopexit}\}$
(I2) a_α and a_β are both (read, var)
(I3) a_α is in $\{(\mathsf{write}, var_\alpha), (\mathsf{read}, var_\alpha)\}$ and a_β is in $\{(\mathsf{write}, var_\beta), (\mathsf{read}, var_\beta)\}$ and $var_\alpha \neq var_\beta$

6.1.4 Checking preemption-safety

Under abstraction, we model each thread as a nondeterministic finite automaton (NFA) over a finite alphabet consisting of abstract observable symbols. This enables us to construct NFAs NP_{abs} and P'_{abs} accepting the languages $[\![\mathscr{C}_{abs}]\!]^{NP}$ and $[\![\mathscr{C}'_{abs}]\!]^{P}$, respectively. $\mathscr{C}_{abs}$ is the abstract program corresponding to the input program $\mathscr{C}$ and $\mathscr{C}'_{abs}$ is the program corresponding to the result of the synthesis $\mathscr{C}'$. It turns out that preemption-safety of $\mathscr{C}'$ w.r.t. $\mathscr{C}$ is implied by preemption-safety of $\mathscr{C}'_{abs}$ w.r.t. $\mathscr{C}_{abs}$, which, in turn, is implied by *language inclusion modulo* I_D of NFAs P'_{abs} and NP_{abs}. NFAs P'_{abs} and NP_{abs} satisfy language inclusion modulo I_D if any word accepted by P'_{abs} is equivalent to some word obtainable by repeatedly commuting adjacent independent symbol pairs in a word accepted by NP_{abs}.

Proposition 1 *Given concurrent programs* $\mathscr{C}$ *and* $\mathscr{C}'$, $[\![\mathscr{C}'_{abs}]\!]^{P} \Subset_{abs} [\![\mathscr{C}_{abs}]\!]^{NP}$ *iff* $\mathscr{L}(\mathsf{P}'_{abs}) \subseteq \mathrm{Clo}_{I_D}(\mathscr{L}(\mathsf{NP}_{abs}))$.

Proof By construction P'_{abs}, resp. NP_{abs}, accept exactly the observation sequences that $\mathscr{C}'_{abs}$, resp. $\mathscr{C}_{abs}$, may produce under the preemptive, resp. non-preemptive, semantics (denoted by $[\![\mathscr{C}'_{abs}]\!]^{P}$, resp. $[\![\mathscr{C}_{abs}]\!]^{NP}$). It remains to show that two observation sequences $\omega_1 = \alpha_0 \dots \alpha_k$ and $\omega_2 = \beta_0 \dots \beta_k$ are equivalent iff $\omega_1 \in \mathrm{Clo}_{I_D}(\{\omega_2\})$.

We first show that $\omega_1 \in \mathrm{Clo}_{I_D}(\{\omega_2\})$ implies ω_1 is equivalent to ω_2. The proof proceeds by induction: The base case is that no symbols are swapped and is trivially true. The inductive case assumes that ω' is equivalent to ω_2 and we needs to show that after one single swap operation in ω', resulting in ω'', ω' is equivalent to ω'' and therefore by transitivity also equivalent to ω_2. Rule (*A1*) holds because I_D does not allow symbols of the same thread to be swapped (I0). To prove (*A2*) we use the fact that writes to the same variable cannot be

[2] The equivalence classes of $\approx_I^t$ are Mazurkiewicz traces.

Thread T1

ℓ_1 **while** (*) **do**
ℓ_2 signal(ch-sym) ▷ choose symbol
ℓ_3 wait_reset(ch-sym-compl)
ℓ_4 $\mathtt{sA}^1 \leftarrow \Delta_A^1(\mathtt{sA}^1, \ldots, \mathtt{sA}^n, \tau^1, \ldots, \tau^p)$
ℓ_5 $\ldots$
ℓ_6 $\mathtt{sA}^n \leftarrow \Delta_A^n(\mathtt{sA}^1, \ldots, \mathtt{sA}^n, \tau^1, \ldots, \tau^p)$
ℓ_7 $\mathtt{sB}^1 \leftarrow \Delta_B^1(\mathtt{sB}^1, \ldots, \mathtt{sB}^m, \tau^1, \ldots, \tau^p)$
ℓ_8 $\ldots$
ℓ_9 $\mathtt{sB}^m \leftarrow \Delta_B^m(\mathtt{sB}^1, \ldots, \mathtt{sB}^m, \tau^1, \ldots, \tau^p)$
ℓ_{10} $\mathtt{final} \leftarrow \big(\mathtt{simA} \implies \bigvee_{q \in F_A}(\mathtt{sA}^1 = q^1 \wedge \cdots \wedge \mathtt{sA}^n = q^n)\big) \wedge \big(\neg\mathtt{simA} \implies \bigvee_{q \in F_B}(\mathtt{sB}^1 = q^1 \wedge \cdots \wedge \mathtt{sB}^m = q^m)\big)$
ℓ_{11} assume(final)

Thread T2

ℓ_{12} simA $\leftarrow$ true
ℓ_{13} simA $\leftarrow$ false

Thread $\mathtt{T}_\alpha$

ℓ_{14} **while** (*) **do**
ℓ_{15} wait_reset(ch-sym)
ℓ_{16} $\tau^1 \leftarrow \alpha^1$
ℓ_{17} $\ldots$
ℓ_{18} $\tau^p \leftarrow \alpha^p$
ℓ_{o1} write($v_{\{\alpha,\alpha_1\}}$)
$\ldots$
ℓ_{ok} write($v_{\{\alpha,\alpha_k\}}$)
ℓ_{19} signal(ch-sym-compl)

Fig. 14 Simulator algorithm

swapped (*I2*), (*I3*). To prove (*A3*) we use the fact that reads and writes to the same variable are not independent (*I2*), (*I3*).

It remains to show that ω_1 is equivalent to ω_2 implies $\omega_1 \in \mathrm{Clo}_{I_D}(\{\omega_2\})$. Clearly ω_1 and ω_2 consist of the same multiset of symbols (*A1*). Therefore it is possible to transform ω_2 into ω_1 by swapping adjacent symbols. It remains to show that all swaps involve independent symbols. By (*A1*) the order of events in each thread does not change, therefore condition (I0) is always fulfilled. Branch tags can swap with every other symbol (*I1*) and accesses to different variables can swap with each other (*I3*). For each variables *ShVar* (*A2*) ensures that writes are in the same order and (*A3*) allows reads in between to be reordered. These swaps are allowed by (*I2*). No other swaps can occur. □

6.2 Checking language inclusion

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

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

We now show that this general undecidability result extends to our specific NFAs and independence relation I_D.

Theorem 3 *For NFAs* P'_{abs} *and* NP_{abs} *constructed from* $\mathscr{C}_{abs}$*, the problem* $\mathscr{L}(\mathsf{P}'_{abs}) \subseteq \mathrm{Clo}_{I_D}(\mathscr{L}(\mathsf{NP}_{abs}))$ *is undecidable.*

Proof Our proof is by reduction from the language inclusion modulo an independence relation problem (Definition 1). Theorem 3 follows from the undecidability of this problem (Theorem 2).

Assume we are given NFAs $A = (Q_A, \Sigma, \Delta_A, Q_{\iota,A}, F_A)$ and $B = (Q_B, \Sigma, \Delta_B, Q_{\iota,B}, F_B)$ and an independence relation $I \subseteq \Sigma \times \Sigma$. Without loss of generality we assume A and

B to be deterministic, complete, and free of ϵ-transitions, meaning from every state there is exactly one transition for each symbol. We show that we can construct a program $\mathscr{C}_{abs}$ that is preemption-safe iff $\mathscr{L}(A) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$.

For our reduction we construct a program $\mathscr{C}_{abs}$ that simulates A or B if run with a preemptive scheduler and simulates only B if run with a non-preemptive scheduler. Note that $\mathscr{L}(A) \cup \mathscr{L}(B) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$ iff $\mathscr{L}(A) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$. For every symbol $\alpha \in \Sigma$ our simulator produces a sequence ω_α of abstract observable symbols. We say two such sequences ω_α and ω_β *commute* if $\omega_\alpha \cdot \omega_\beta \approx^t_{I_D} \omega_\beta \cdot \omega_\alpha$, i.e, if $\omega_\beta \cdot \omega_\alpha$ can be obtained from $\omega_\alpha \cdot \omega_\beta$ by repeatedly swapping adjacent symbol pairs in I_D.

We will show that (a) $\mathscr{C}_{abs}$ simulates A or B if run with a preemptive scheduler and simulates only B if run with a non-preemptive scheduler, and (b) sequences ω_α and ω_β commute iff $(\alpha, \beta) \in I$.

The simulator is shown in Fig. 14. States and symbols of A and B are mapped to natural numbers and represented as bitvectors to enable simulation using the language $\mathscr{W}_{abs}$. In particular we use Boolean guard variables from $\mathscr{W}_{abs}$ to represent the bitvectors. We use `true` to represent 1 and `false` to represent 0. As the state space and the alphabet are finite we know the number of bits needed a priori. We use n, m, and p for the number of bits needed to represent Q_A, Q_B, and Σ, respectively. The transition functions Δ_A and Δ_B likewise work on the individual bits. We represent bitvector x of length n as $x^1 \dots x^n$.

Thread `T1` simulates both automata A and B simultaneously. We assume the initial states of A and B are mapped to the number 0. In each iteration of the loop in thread `T1` a symbol $\alpha \in \Sigma$ is chosen non-deterministically and applied to both automata (we discuss this step in the next paragraph). Whether thread `T1` simulates A or B is decided only in the end: depending on the value of `simA` we assert that a final state of A or B was reached. The value of `simA` is assigned in thread `T2` and can only be `true` if `T2` is preempted between locations ℓ_{12} and ℓ_{13}. With the non-preemptive scheduler the variable `simA` will always be `false` because thread `T2` cannot be preempted. The simulator can only reach the $\langle$`terminated`$\rangle$ state if all assumptions hold as otherwise it would end in the $\langle$`failed`$\rangle$ state. The guard `final` will only be assigned `true` in ℓ_{10} if either `simA` is `false` and a final state of B has been reached or if `simA` is `true` and a final state of A has been reached. Therefore the valid non-preemptive executions can only simulate B. In the preemptive setting the simulator can simulate either A or B because `simA` can be either `true` or `false`. Note that the statement in location ℓ_{10} executes atomically and the value of `simA` cannot change during its evaluation. This means that P'_{abs} simulates $\mathscr{L}(A) \cup \mathscr{L}(B)$ and NP_{abs} simulates $\mathscr{L}(B)$.

We use τ to store the symbol used by the transition function. The choice of the next symbol needs to be non-deterministic to enable simulation of A, B and there is no havoc statement in $\mathscr{W}_{abs}$. We therefore use the fact that the next thread to execute is chosen non-deterministically at a preemption point. We define a thread $\mathtt{T}_\alpha$ for every $\alpha \in \Sigma$ that assigns to τ the number α maps to. Threads $\mathtt{T}_\alpha$ can only run if the conditional variable `ch-sym` is set to 1 by the notify statement in ℓ_2. The wait_reset(`ch-sym-compl`) in ℓ_3 is a preemption point for the non-preemptive semantics. Then, exactly one thread $\mathtt{T}_\alpha$ can proceed because the wait_reset(`ch-sym`) statement in ℓ_{15} atomically resets `ch-sym` to 0. After setting τ and outputting the representation of α thread $\mathtt{T}_\alpha$, notifies thread `T1` using condition variable `ch-sym-compl`. Another symbol can only be produced in the next loop iteration of `T1`.

To produce an observable sequence faithful to I for each symbol in Σ we define a homomorphism h that maps symbols from Σ to sequences of observables. Assuming the symbol $\alpha \in \Sigma$ is chosen, we produce the following observables:

- *Loop tag* To output α the thread $\mathtt{T}_\alpha$ has to perform one loop iteration. This implicitly produces a loop tag $(\mathtt{T}_\alpha, \mathsf{loop}, \ell_{14})$.
- *Conflict variables* For each pair of $(\alpha, \alpha_i) \notin I$, we define a conflict variable $v_{\{\alpha,\alpha_i\}}$. Note that $v_{\{\alpha,\alpha_i\}} = v_{\{\alpha_i,\alpha\}}$ and two writes to $v_{\{\alpha,\alpha_i\}}$ do not commute under I_D. For each α_i, we produce a tag $(T_\alpha, (\mathsf{write}, v_{\{\alpha,\alpha_i\}}, \ell_{oi}))$. Therefore if two variables α_1 and α_2 are dependent the observation sequences produced for each of them will contain a write to $v_{\{\alpha_1,\alpha_2\}}$.

Formally, the homomorphism h is given by $h(\alpha) = (\mathtt{T}_\alpha, \mathsf{loop}, \ell_{14}); (\mathtt{T}_\alpha, (\mathsf{write}, v_{\{\alpha,\alpha_1\}}), \ell_{o1}); \cdots ; (\mathtt{T}_\alpha, (\mathsf{write}, v_{\{\alpha,\alpha_k\}}), \ell_{ok})$. For a sequence $\sigma = \alpha_1 \ldots \alpha_n$ use define $h(\sigma) = h(\alpha_1) \ldots h(\alpha_n)$.

We show that $(\alpha_1, \alpha_2) \in I$ iff $h(\alpha_1)$ and $h(\alpha_2)$ commute. The loop tags are independent iff $\alpha_1 \neq \alpha_2$. If $\alpha_1 = \alpha_2$ then $(\alpha_1, \alpha_2) \notin I$ and $h(\alpha_1)$ and $h(\alpha_2)$ do not commute due to the loop tags. Assuming $(\alpha_1, \alpha_2) \in I$ then $h(\alpha_1)$ and $h(\alpha_2)$ commute because they have no common conflict variable they write to. On the other hand, if $(\alpha_1, \alpha_2) \notin I$, then both $h(\alpha_1)$ and $h(\alpha_2)$ will contain $(\mathtt{T}_{\alpha_{\{1,2\}}}, (\mathsf{write}, v_{\{\alpha_1,\alpha_2\}}), \ell_{oi})$ and therefore cannot commute. We extend this result to sequences and have that $h(\sigma') \approx^t_{I_D} h(\sigma)$ iff $\sigma' \approx^t_I \sigma$.

This concludes our reduction. It remains to show that $\mathscr{C}_{abs}$ is preemption-safe iff $\mathscr{L}(A) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$. By Proposition 1 it suffices to show that $\mathscr{L}(A) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$ iff $\mathscr{L}(\mathsf{P}'_{abs}) \subseteq \mathrm{Clo}_{I_D}(\mathscr{L}(\mathsf{NP}_{abs}))$.

1. We assume that $\mathscr{L}(A) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$. Then, for every word $\sigma \in \mathscr{L}(A)$ we have that $\sigma \in \mathrm{Clo}_I(\mathscr{L}(B))$. By construction $h(\sigma) \in \mathscr{L}(\mathsf{P}'_{abs})$. It remains to show that $h(\sigma) \in \mathrm{Clo}_{I_D}(\mathscr{L}(\mathsf{NP}_{abs}))$. By $\sigma \in \mathrm{Clo}_I(\mathscr{L}(B))$ we know there exists a word $\sigma' \in \mathscr{L}(B)$, such that $\sigma' \approx^t_I \sigma$. Therefore also $h(\sigma') \approx^t_{I_D} h(\sigma)$ and by construction $h(\sigma') \in \mathscr{L}(\mathsf{NP}_{abs})$.
2. We assume that $\mathscr{L}(A) \nsubseteq \mathrm{Clo}_I(\mathscr{L}(B))$. Then, there exists a word $\sigma \in \mathscr{L}(A)$ such that $\sigma \notin \mathrm{Clo}_I(\mathscr{L}(B))$. By construction $h(\sigma) \in \mathscr{L}(\mathsf{P}'_{abs})$. Let us assume towards contradiction that $h(\sigma) \in \mathrm{Clo}_{I_D}(\mathscr{L}(\mathsf{NP}_{abs}))$. Then there exists a word ω in $\mathscr{L}(\mathsf{NP}_{abs})$ such that $\omega \approx^t_{I_D} h(\sigma)$. By construction, this implies there exists some $\sigma' \in \mathscr{L}(B)$ such that $\omega = h(\sigma')$ and $h(\sigma') \approx^t_{I_D} h(\sigma)$. Thus, there exists $\sigma' \in \mathscr{L}(B)$ such that $\sigma' \approx^t_I \sigma$. This implies $\sigma \in \mathrm{Clo}_I(\mathscr{L}(B))$, which is a contradiction. □

Fortunately, a bounded version of the language inclusion modulo I problem is decidable. Recall the relation $\approx_I$ over Σ^* from Sect. 6.1. We define a symmetric binary relation $\approx^i_I$ over Σ^*: $(\sigma, \sigma') \in \approx^i_I$ iff $\exists(\alpha, \beta) \in I$: $(\sigma, \sigma') \in \approx_I, \sigma[i] = \sigma'[i+1] = \alpha$ and $\sigma[i+1] = \sigma'[i] = \beta$. Thus $\approx^i_I$ consists of all words that can be obtained 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 $\exists \sigma_1, \ldots, \sigma_t$: $(\sigma, \sigma_1) \in \approx^{i_1}_I, \ldots, (\sigma_t, \sigma') \in \approx^{i_{t+1}}_I$ 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. We recursively define $\asymp^k$ as follows: $\asymp^0$ is the identity relation id. For $k > 0$ we define $\asymp^k = \asymp \circ \asymp^{k-1}$, the composition of $\asymp$ with $\asymp^{k-1}$. Given a language $\mathscr{L}$ over Σ, we use $\mathrm{Clo}_{k,I}(\mathscr{L})$ to denote the set $\{\sigma \in \Sigma^* : \exists \sigma' \in \mathscr{L}$ with $(\sigma, \sigma') \in \asymp \}$. In other words, $\mathrm{Clo}_{k,I}(\mathscr{L})$ consists of all words which can be generated from $\mathscr{L}$ using a finite-state transducer that remembers at most k symbols of its input words in its states. By definition we have $\mathrm{Clo}_{0,I}(\mathscr{L}) = \mathscr{L}$.

Example 1 We assume the language $\mathscr{L} = \{a, b\}^*$, where $(a, b) \in I$.

- $aaab \asymp^1_I aaba$ because one can swap the letters as position 3 and 4.
- $aaab \not\asymp^1_I abaa$ because one can only swap the letters as position 3 and 4 in one pass, but not after that swap 2 and 3.

- However, *aaab* $\asymp_I^2$ *abaa*, as two passes suffice to do the two swaps.
- *baaa* $\asymp_I^1$ *aaba* because in a single pass one can swap 1 and 2 and then 2 and 3.

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

Theorem 4 *For NFAs A, B over $\Sigma, I \subseteq \Sigma \times \Sigma$ and a constant $k \geq 0, \mathscr{L}(A) \subseteq \mathrm{Clo}_{k,I}(\mathscr{L}(B))$ is decidable.*

We present an algorithm to check k-bounded language inclusion modulo I, based on the antichain algorithm for standard language inclusion [11].

6.3 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 $\mathscr{L}(A) \subseteq \mathscr{L}(B)$ 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. Without loss of generality we assume that A and B do not have ϵ-transitions. 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 $\mathsf{succ}_\alpha(s_A)$ denote $\{s'_A \in Q_A : (s_A, \alpha, s'_A) \in \Delta_A\}$. Given set of states $S_B \subseteq Q_B$, let $\mathsf{succ}_\alpha(S_B)$ 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 $\mathsf{succ}_\alpha(s_A, S_B)$ denote $\{(s'_A, S'_B) : s'_A \in \mathsf{succ}_\alpha(s_A), S'_B = \mathsf{succ}_\alpha(S_B)\}$.

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 $S_B \cap F_B = \emptyset$, 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', if there does not exist a tuple p in the antichain with $p \sqsubseteq p'$, then p' is added to the frontier set or antichain (*Rule R1*). If p' is added and 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 (*Rule R2*). The algorithm terminates by either reporting a counterexample, or by declaring success when the frontier becomes empty.

6.4 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 $\mathrm{Clo}_{k,I}(\mathscr{L}(\mathrm{B}))$. 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 words over Σ of up to k length. Intuitively, the words η_1 and η_2 store symbols that are expected to be matched later along a run. The word η_1 contains a list of symbols for transitions taken by $B_{k,I}$, but not yet matched in B, whereas η_2 contains a list of symbols for transitions taken in B, but not yet matched in $B_{k,I}$. We use $\emptyset$ to denote the empty list. Since for every transition of $B_{k,I}$, the automaton B will perform one transition, we have $|\eta_1| = |\eta_2|$. The set of initial states of $B_{k,I}$ is $\{(s_B, \emptyset, \emptyset) : s_B \in Q_{\iota,B}\}$. The set of final states of $B_{k,I}$ is $\{(s_B, \emptyset, \emptyset) : s_B \in F_B\}$. The transition relation $\Delta_{B_{k,I}}$ is constructed by repeatedly performing the following steps, in

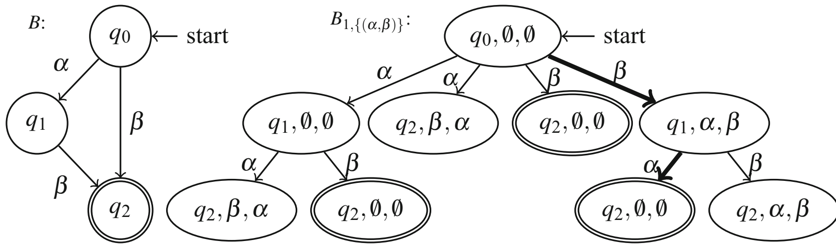

Fig. 15 Example for illustrating construction of $B_{k,I}$ for $k = 1$ and $I = \{(\alpha, \beta)\}$

order, for each state (s_B, η_1, η_2) and each symbol α. In what follows, $\eta[\backslash i]$ denotes the word obtained from η by removing its ith symbol.
Given (s_B, η_1, η_2) and α

- *Step S1* Pick *new* s'_B and $\beta \in \Sigma$ such that $(s_B, \beta, s'_B) \in \Delta_B$
- *Step S2*
  - (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, $\eta'_1 := \eta_1[\backslash i]$ and $\eta'_2 := \eta_2$
  - (c) else, go to S1
- *Step S3*
  - (a) If $\forall i$: $\eta'_2[i] \neq \beta$ and β is independent of all symbols in η'_2, $\eta''_1 := \eta'_1 \cdot \beta$ and $\eta''_2 := \eta'_2$,
  - (b) else, if $\exists i$: $\eta'_2[i] = \beta$ and β is independent of all symbols in η'_2 prior to i, $\eta'_2 := \eta'_2[\backslash i]$ and $\eta''_1 := \eta'_1$
  - (c) else, go to S1
- *Step S4* Add $((s_B, \eta_1, \eta_2), \alpha, (s'_B, \eta''_1, \eta''_2))$ to $\Delta_{B_{k,I}}$ and go to 1.

Example 2 In Fig. 15, we have an NFA B with $\mathscr{L}(\mathrm{B}) = \{\alpha\beta, \beta\}$, $I = \{(\alpha, \beta)\}$ and $k = 1$. The states of $B_{k,I}$ are triples (q, η_1, η_2), where $q \in Q_B$ and $\eta_1, \eta_2 \in \{\alpha, \beta\}^*$. We explain the derivation of a couple of transitions of $B_{k,I}$. The transition shown in bold from $(q_0, \emptyset, \emptyset)$ on symbol β is obtained by applying the following steps once: S1. Pick q_1 following the transition $(q_0, \alpha, q_1) \in \Delta_B$. S2(a). $\eta'_2 := \beta, \eta'_1 := \emptyset$. S3(a). $\eta''_1 := \alpha, \eta''_2 := \beta$. S4. Add $((q_0, \emptyset, \emptyset), \beta, (q_1, \alpha, \beta))$ to $\Delta_{B_{k,I}}$. The transition shown in bold from (q_1, α, β) on symbol α is obtained as follows: S1. Pick q_2 following the transition $(q_1, \beta, q_2) \in \Delta_B$. S2(b). $\eta'_1 := \emptyset, \eta'_2 := \beta$. S3(b). $\eta''_2 := \emptyset, \eta''_1 := \emptyset$. S4. Add $((q_1, \alpha, \beta), \beta, (q_2, \emptyset, \emptyset))$ to $\Delta_{B_{k,I}}$. It can be seen that $B_{k,I}$ accepts the language $\{\alpha\beta, \beta\alpha, \beta\} = \mathrm{Clo}_{k,I}(\mathscr{L}(\mathrm{B}))$.

Proposition 2 *Given $k \geq 0$, the automaton $B_{k,I}$ accepts at least* $\mathrm{Clo}_{k,I}(\mathscr{L}(B))$.

Proof The proof is by induction on k. The base case is trivially true, as $\mathscr{L}(B_{0,I}) = \mathscr{L}(B) = \mathrm{Clo}_{0,I}(\mathscr{L}(B))$. The induction case assumes that $B_{k,I}$ accepts at least $\mathrm{Clo}_{k,I}(\mathscr{L}(\mathrm{B}))$ and we want to show that $B_{k+1,I}$ accepts at least $\mathrm{Clo}_{k+1,I}(\mathscr{L}(B))$. We take a word $\omega \in \mathrm{Clo}_{k+1,I}(\mathscr{L}(B))$. It must be derived from a word $\omega' \in \mathrm{Clo}_{k,I}(\mathscr{L}(B))$ by one additional forward pass of swapping. $B_{k+1,I}$ accepts ω: In step S1 we pick the same transitions in Δ_B as to accept ω'. Steps S2 and S3 will be identical as for ω' with the exception of those

adjacent symbol pairs that are newly swapped in ω. For those pairs the symbols are first added to η_2 and η_1 by S2 and S3. In the next step they are removed because the swapping only allows adjacent symbols to be swapped. This also shows that the bound $k+1$ suffices to accept ω. □

In general NFA $B_{k,I}$ can accept words not in $\text{Clo}_{k,I}(\mathscr{L}(\text{B}))$. Intuitively this is because $B_{k,I}$ has two stacks and can also accept words where the swapping is done in a backward pass (instead of a forward pass required in our definition). For our purposes it is sound to accept more words as long as they are obtained only by swapping independent symbols.

Proposition 3 *Given $k \geq 0$, the automaton $B_{k,I}$ accepts at most* $\text{Clo}_I(\mathscr{L}(B))$.

Proof We need to show that $\omega' \in B_{k,I} \implies \omega' \in \text{Clo}_I(\mathscr{L}(B))$. For this we need to show that ω' is a permutation of a word $\omega \in \mathscr{L}(B)$ by repeatedly swapping independent, adjacent symbols. The word ω' must be a permutation of ω because $B_{k,I}$ only accepts if η_1 and η_2 are empty and the stacks represent exactly the symbols not matched yet in NFA B. Further, we need to show only independent symbols may be swapped. The stack η_1 contains the symbols not yet matched by B and η_2 the symbols that were instead accepted by B, but not yet presented as input to $B_{k,I}$. Before adding a new symbol to the stack we ensure it is independent with all symbols on the other stack because once matched later it will have to come after all of these. When a symbols is removed it is ensured that it is independent with all symbols on its own stack because it is practically moved ahead of the other symbols on the stack. □

6.5 Language inclusion check algorithm

We develop a procedure to check language inclusion modulo I (Sect. 6.4) by iteratively increasing the bound k. 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.

The algorithm for k-bounded language inclusion modulo I is presented as function INCLUSION in Algorithm 1 (ignore Lines 22–25 for now). The antichain set consists of tuples of the form $(s_A, S_{B_{k,I}})$, where $s_A \in Q_A$ and $S_{B_{k,I}} \subseteq Q_B \times \Sigma^k \times \Sigma^k$. The frontier consists of tuples of the form $(s_A, S_{B_{k,I}}, cex)$, where $cex \in \Sigma^*$. The word cex is a sequence of symbols of transitions explored in A to get to state s_A. If the language inclusion check fails, cex is returned as a counterexample to language inclusion modulo I. 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 and returns the counterexample cex (Line 18). If this check succeeds, the successors are computed (Line 20). If a successor satisfies rule R1, it is ignored (Line 21), otherwise it is added to the frontier (Line 26) and the antichain (Line 27). When adding a successor to the frontier the symbol α it appended to the counterexample, denoted as $cex \cdot \alpha$. During the update of the antichain the algorithm ensures that its invariant is preserved according to rule R2.

We need to ensure that our language inclusion honors the bound k by ignoring states that exceed the bound. These states are stored for later to allow for a restart of the language inclusion algorithm with a higher bound. 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 then prune $S'_{B_{k,I}}$ by removing all states (s_B, η_1, η_2) where $|\eta_1| > k \vee |\eta_2| > k$ (line

Algorithm 1 Checking language inclusion modulo I

Require: Automata $A = (Q_A, \Sigma, \Delta_A, Q_{\iota,A}, F_A)$, $B = (Q_B, \Sigma, \Delta_B, Q_{\iota,B}, F_B)$ and independence relation $I \subseteq \Sigma \times \Sigma$
Ensure: `true` iff $\mathscr{L}(A) \subseteq \mathrm{Clo}_I(\mathscr{L}(B))$

1 $\mathit{frontier} \leftarrow \{(s_A, \{(Q_{\iota,B}, \emptyset, \emptyset)\}, \emptyset) : s_A \in Q_{\iota,A}\}$
2 No tuple in *frontier* is *dirty*
3 $\mathit{antichain} \leftarrow \mathit{frontier}$
4 $\mathit{overflow} \leftarrow \emptyset$
5 $k \leftarrow 2$
6 **while** `true` **do**
7 $\quad \mathit{cex} \leftarrow$ INCLUSION(k)
8 $\quad$ **if** $\mathit{cex} \neq$ `true` $\wedge$ cex is spurious **then**
9 $\quad\quad k \leftarrow k + 1$
10 $\quad\quad \mathit{frontier} \leftarrow \{(s_A, S_{B_{k,I}}) \in \mathit{frontier} : S_{B_{k,I}}$ not *dirty*$\} \cup \mathit{overflow}$
11 $\quad\quad \mathit{antichain} \leftarrow \{(s_A, S_{B_{k,I}}) \in \mathit{antichain} : S_{B_{k,I}}$ not *dirty*$\} \cup \mathit{overflow}$
12 $\quad\quad \mathit{overflow} \leftarrow \emptyset$
13 $\quad$ **else**
14 $\quad\quad$ **return** *cex*

15 **function** INCLUSION(k)
16 $\quad$ **while** $\mathit{frontier} \neq \emptyset$ **do**
17 $\quad\quad$ remove a tuple $(s_A, S_{B_{k,I}}, \mathit{cex})$ from *frontier*
18 $\quad\quad$ **if** $s_A \in F_A \wedge (S_{B_{k,I}} \cap F_B) = \emptyset$ **then return** *cex*
19 $\quad\quad$ **for all** $\alpha \in \Sigma$ **do**
20 $\quad\quad\quad (s'_A, S'_{B_{k,I}}) \leftarrow \mathsf{succ}_\alpha(s_A, S_{B_{k,I}})$
21 $\quad\quad\quad$ **if** $\nexists p \in \mathit{antichain} : p \sqsubseteq (s'_A, S'_{B_{k,I}})$ **then** ▷ Rule R1
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 $\quad\quad\quad\quad \mathit{antichain} \leftarrow (\mathit{antichain} \setminus \{p : (s'_A, S'_{B_{k,I}}) \sqsubseteq p\}) \cup \{(s'_A, S'_{B_{k,I}})\}$ ▷ Rule R2
28 $\quad$ **return** `true`

24) and mark $S'_{B_{k,I}}$ as *dirty* (line 24). If we find a counterexample to language inclusion we return it and test if it is spurious (Line 8). In case it is spurious 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 set (Line 12) to the antichain set and frontier. Intuitively this will undo all exploration from the point(s) the bound was exceeded and restarts from that/those point(s).

We call a counterexample *cex* from our language inclusion procedure spurious if it is not a counterexample to the unbounded language inclusion, formally $\mathit{cex} \in \mathrm{Clo}_I(\mathscr{L}(B))$. This test is decidable because there is only a finite number of permutations of *cex*. This spuriousness arises from the fact that the bounded language-inclusion algorithm is incomplete and every spurious example can be eliminated by sufficiently increasing the bound k. Note, however,

that there exists automata and independence relations for which there is a (different) spurious counterexample for every k. In practice we test if a *cex* is spurious by building an automata A that accepts exactly *cex* and running the language inclusion algorithm with k being the length of *cex*. This is very fast because there is exactly one path through A.

Theorem 5 (bounded language inclusion check) *The procedure* INCLUSION *of Algorithm 1 decides $\mathscr{L}(A) \subseteq \mathscr{L}(B_{k,I})$ for NFAs A, B, bound k, and independence relation I.*

Proof Our algorithm takes as arguments automata A and B. Conceptually, the algorithm constructs $B_{k,I}$ and uses the antichain algorithm [11] to decide the language inclusion. For efficiency, we modify the original antichain language inclusion algorithm to construct the automaton B_I on the fly in the successor relation succ (line 20). The bound k is enforced separately in line 22. □

Theorem 6 (preemption-safety problem) *If program $\mathscr{C}$ is not preemption-safe ($[\![\mathscr{C}]\!]^P \not\subseteq [\![\mathscr{C}]\!]^{NP}$), then Algorithm 1 will return* `false`.

Proof By Theorem 1 we know $[\![\mathscr{C}_{abs}]\!]^P \not\subseteq_{abs} [\![\mathscr{C}_{abs}]\!]^{NP}$. From Proposition 1 we get $\mathscr{L}(\mathsf{P}_{abs}) \nsubseteq \mathrm{Clo}_{I_D}(\mathscr{L}(\mathsf{NP}_{abs}))$. From Proposition 3 we know that for any k this is equivalent to $\mathscr{L}(\mathsf{P}_{abs}) \nsubseteq \mathscr{L}(B_{k,I})$, where $B = \mathsf{NP}_{abs}$. Theorem 5 shows that Algorithm 1 decides this for any bound k. □

7 Finding and Enforcing Mutex Constraints in P'_{abs}

If the language inclusion check fails it returns a counterexample trace. Using this counterexample we derive a set of *mutual exclusion (mutex) constraints* that we enforce in P'_{abs} to eliminate the counterexample and then rerun the language inclusion check with the new P'_{abs}.

7.1 Finding mutex constraints

The counterexample *cex* returned by the language inclusion check is a sequence of observables. Since our observables record every branching decision it is easy to reconstruct from *cex* a sequence of event identifiers: $tid_0.\ell_0; \ldots; tid_n.\ell_n$, where each ℓ_i is a location identifier from $\mathscr{C}_{abs}$. In this section we use *cex* to refer to such sequences of event identifiers. We define the *neighborhood* of *cex*, denoted *nhood*(*cex*), as the set of all traces that are permutations of the events in *cex* and preserve the order of events from the same thread. We separate traces in *nhood*(*cex*) into *good* and *bad traces*. Good traces are all traces that are infeasible under the non-preemptive semantics or that produce an observation sequence that is equivalent to that of a trace feasible under the non-preemptive semantics. All remaining traces in *nhood*(*cex*) are bad. The goal of our counterexample analysis is to characterize all bad traces in *nhood*(*cex*) in order to enable inference of mutex constraints.

In order to succinctly represent subsets of *nhood*(*cex*), we use *ordering constraints* between events expressed as *happens-before formulas* (*HB-formulas*) [15]. Intuitively, ordering constraints are of the following forms: (a) atomic ordering constraints $\varphi = A < B$ 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 $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2$ and $\neg\varphi_1$. We have that $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ respectively represent the intersection and union of the set of traces represented by φ_1 and φ_2, and that $\neg\varphi_1$ represents the complement (with respect to *nhood*(*cex*)) of the traces represented by φ_1.

7.1.1 Non-preemptive neighborhood

First, we define function Φ to extract a conjunction of atomic ordering constraints from a trace π, such that all traces π' in $\Phi(\pi)$ produce an observation sequence equivalent to π. Then, we obtain a correctness constraint φ that represents all good traces in *nhood*(*cex*). Remember, that the good traces are those that are observationally equivalent to a non-preemptive trace. The correctness constraint φ is a disjunction over the ordering constraints from all traces in *nhood*(*cex*) that are feasible under non-preemptive semantics: $\varphi_G = \bigvee_{\pi \in \text{non-preemptive}} \Phi(\pi)$. $\Phi(\pi)$ enforces the order between conflicting accesses in the abstract trace π:

$$\Phi(\pi) = \bigwedge\{\mathtt{T}i.\ell_j < \mathtt{T}k.\ell_l : i \neq k \wedge \mathtt{T}i.\ell_j \text{ precedes } \mathtt{T}k.\ell_l \text{ in } \pi \wedge \\ \mathtt{T}i.\ell_j, \mathtt{T}k.\ell_l \text{ access same variable} \wedge \mathtt{T}i.\ell_j \text{ or } \mathtt{T}k.\ell_l \text{ is a write}\}$$

Example Recall the counterexample trace from the running example in Sect. 3: $cex = \mathtt{T1}.\ell_{1a};\ \mathtt{T1}.\ell_{1b};\ \mathtt{T2}.\ell_{1a};\ \mathtt{T2}.\ell_{1b};\ \mathtt{T1}.\ell_2;\ \mathtt{T2}.\ell_2;\ \mathtt{T2}.\ell_{3a};\ \mathtt{T2}.\ell_{3b};\ \mathtt{T2}.\ell_4;\ \mathtt{T1}.\ell_{3a};\ \mathtt{T1}.\ell_{3b};\ \mathtt{T1}.\ell_4$. There are two traces in *nhood*(*cex*) that are feasible under non-preemptive semantics:

- $\pi_1 = \mathtt{T1}.\ell_{1a};\ \mathtt{T1}.\ell_{1b};\ \mathtt{T1}.\ell_2;\ \mathtt{T1}.\ell_{3a};\ \mathtt{T1}.\ell_{3b};\ \mathtt{T1}.\ell_4;\ \mathtt{T2}.\ell_{1a};\ \mathtt{T2}.\ell_{1b};\ \mathtt{T2}.\ell_2;\ \mathtt{T2}.\ell_{3a};\ \mathtt{T2}.\ell_{3b};\ \mathtt{T2}.\ell_4$ and
- $\pi_2 = \mathtt{T2}.\ell_{1a};\ \mathtt{T2}.\ell_{1b};\ \mathtt{T2}.\ell_2;\ \mathtt{T2}.\ell_{3a};\ \mathtt{T2}.\ell_{3b};\ \mathtt{T2}.\ell_4;\ \mathtt{T1}.\ell_{1a};\ \mathtt{T1}.\ell_{1b};\ \mathtt{T1}.\ell_2;\ \mathtt{T1}.\ell_{3a};\ \mathtt{T1}.\ell_{3b};\ \mathtt{T1}.\ell_4$.

We represent

- π_1 as $\Phi(\pi_1) = (\{\mathtt{T1}.\ell_{1a}, \mathtt{T1}.\ell_{3a}, \mathtt{T1}.\ell_{3b}\} < \mathtt{T2}.\ell_{3b}) \wedge (\mathtt{T1}.\ell_{3b} < \{\mathtt{T2}.\ell_{1a}, \mathtt{T2}.\ell_{3a}, \mathtt{T2}.\ell_{3b}\}) \wedge (\mathtt{T1}.\ell_2 < \mathtt{T2}.\ell_2)$ and
- π_2 as $\Phi(\pi_2) = (\mathtt{T2}.\ell_{3b} < \{\mathtt{T1}.\ell_{1a}, \mathtt{T1}.\ell_{3a}, \mathtt{T1}.\ell_{3b}\}) \wedge (\{\mathtt{T2}.\ell_{1a}, \mathtt{T2}.\ell_{3a}, \mathtt{T2}.\ell_{3b}\} < \mathtt{T1}.\ell_{3b}) \wedge (\mathtt{T2}.\ell_2 < \mathtt{T1}.\ell_2)$.

The correctness specification is $\varphi_G = \Phi(\pi_1) \vee \Phi(\pi_2)$.

7.1.2 Counterexample enumeration and generalization

We next build a quantifier-free first-order formula Ψ_B over the event identifiers in *cex* such that any model of Ψ_B corresponds to a bad, feasible trace in *nhood*(*cex*). A trace is feasible if it respects the preexisting synchronization, which is not abstracted away. Bad traces are those that are feasible under the preemptive semantics and not in φ_G. Further, we define a generalization function G that works on conjunctions of atomic ordering constraints φ by iteratively removing a constraint as long as the intersection of traces represented by $G(\varphi)$ and φ_G is empty. This results in a local minimum of atomic ordering constraints in $G(\varphi)$, so that removing any remaining constraint would include a good trace in $G(\varphi)$. We iteratively enumerate models ψ of Ψ_B, building a constraint $\varphi_{B'} = \Phi(\psi)$ for each model ψ and generalizing $\varphi_{B'}$ to represent a larger set of bad traces using G. This results in an ordering constraint in disjunctive normal form $\varphi_B = \bigvee_{\psi \in \Psi_B} G(\Phi(\psi))$, such that the intersection of φ_B and φ_G is empty and the union equals *nhood*(*cex*).

Algorithm 2 shows how the algorithm works. For each model ψ of Ψ_B a trace σ is extracted in Line 6. From the trace the formula $\varphi_{B'}$ is extracted using Φ described above (Line 8). Line 10 describes the generalization function G, which is implemented using an unsat core computation. We construct a formula $\varphi_{B'} \wedge \Psi \wedge \varphi_G$, where $\Psi \wedge \varphi_G$ is a hard constraint and φ'_B are soft constraints. A satisfying assignment to this formula models feasible traces that are observationally equivalent to a non-preemptive trace. Since σ is a bad trace

Algorithm 2 Counterexample enumeration and generalization algorithm

Require: Trace π, formula of good traces φ_G in *nhood*(π)
Ensure: HB-formula of bad traces φ_B
1: $\Psi \leftarrow$ quantifier-free first-order formula representing all feasible traces in *nhood*(π)
2: $\Psi_B \leftarrow \Psi \wedge \neg\varphi_G$
3: $\varphi_B \leftarrow$ `false`
4: **while** $\Psi_B \wedge \neg\varphi_B$ is satisfiable **do**
5: $\psi \leftarrow$ satisfying assignment for $\Psi_B \wedge \neg\varphi_B$
6: $\sigma \leftarrow$ trace represented by ψ
7: ▷ Conflicting access analysis
8: $\varphi_{B'} \leftarrow \Phi(\sigma)$
9: ▷ Unsat-core computation
10: $\varphi_{B''} \leftarrow MinUNSATCore(Soft \leftarrow \varphi_{B'},$
11: $Hard \leftarrow \Psi \wedge \varphi_G)$
12: $\varphi_B \leftarrow \varphi_B \vee \varphi_{B''}$
13: **return** φ_B

the formula $\varphi_{B'} \wedge \Psi \wedge \varphi_G$ must be unsatisfiable. The result of the unsat core computation is a formula $\varphi_{B''}$ that is a conjunction of a minimal set of happens-before constraints required to ensure all trace represented by $\varphi_{B''}$ are bad.

Example Our trace *cex* from Sect. 3 is generalized to $G(\Phi(cex)) = \texttt{T2}.\ell_{1a} < \texttt{T1}.\ell_{3b} \wedge \texttt{T1}.\ell_{3b} < \texttt{T2}.\ell_{3b}$. This constraint captures the interleavings where T2 interrupts T1 between locations ℓ_{1a} and ℓ_{3b}. Any trace that fulfills this constraint is bad. All bad traces in *nhood*(cex) are represented as $\varphi_B = (\texttt{T2}.\ell_{1a} < \texttt{T1}.\ell_{3b} \wedge \texttt{T1}.\ell_{3b} < \texttt{T2}.\ell_{3b}) \vee (\texttt{T1}.\ell_{1a} < \texttt{T2}.\ell_{3b} \wedge \texttt{T2}.\ell_{3b} < \texttt{T1}.\ell_{3b})$.

7.1.3 Inferring mutex constraints

From each clause φ in φ_B described above, we infer mutex constraints to eliminate all bad traces satisfying φ. The key observation we exploit is that atomicity violations show up in our formulas as two simple patterns of ordering constraints between events.

1. The first pattern $tid_1.\ell_1 < tid_2.\ell_2 \wedge tid_2.\ell_2' < tid_1.\ell_1'$ (visualized in Fig. 16a) indicates an atomicity violation (thread tid_2 interrupts tid_1 at a critical moment).
2. The second pattern is $tid_1.\ell_1 < tid_2.\ell_2' \wedge tid_2.\ell_2 < tid_1.\ell_1'$ (visualized in Fig. 16b). This pattern is a generalization of the first pattern in that either tid_1 interrupts tid_2 or the other way round.

For both patterns the corresponding mutex constraint is $\mathrm{mtx}(tid_1.[\ell_1{:}\ell_1'], tid_2.[\ell_2{:}\ell_2'])$.

Example The generalized counterexample constraint $\texttt{T2}.\ell_{1a} < \texttt{T1}.\ell_{3b} \wedge \texttt{T1}.\ell_{3b} < \texttt{T2}.\ell_{3b}$ yields the constraint mutex $\mathrm{mtx}(\texttt{T2}.[\ell_{1a}{:}\ell_{3b}], \texttt{T1}.[\ell_{3b}{:}\ell_{3b}])$. In the next section we show how this mutex constraint is enforced in P'_{abs}.

7.2 Enforcing mutex constraints

To enforce mutex constraints in P'_{abs}, we prune paths in P'_{abs} that violate the mutex constraints.

Fig. 16 Atomicity violation patterns

7.2.1 Conflicts

Given a mutex constraint $\text{mtx}(tid_i.[\ell_1{:}\ell'_1], tid_j.[\ell_2{:}\ell'_2])$, a *conflict* is a tuple $(\ell_i^{\text{pre}}, \ell_i^{\text{mid}}, \ell_i^{\text{post}}, \ell_j^{\text{cpre}}, \ell_j^{\text{cpost}})$ of location identifiers satisfying the following:

(a) $\ell_i^{\text{pre}}, \ell_i^{\text{mid}}, \ell_i^{\text{post}}$ are adjacent locations in thread tid_i,
(b) $\ell_j^{\text{cpre}}, \ell_j^{\text{cpost}}$ are adjacent locations in the other thread tid_j,
(c) $\ell_1 \leq \ell_i^{\text{pre}}, \ell_i^{\text{mid}}, \ell_i^{\text{post}} \leq \ell'_1$ and
(d) $\ell_2 \leq \ell_j^{\text{cpre}}, \ell_j^{\text{cpost}} \leq \ell'_2$.

Intuitively, a conflict represents a *minimal violation* of a mutex constraint due to the execution of the statement at location ℓ_j^{cpre} in thread j between the two statements at locations ℓ_i^{pre} and ℓ_i^{mid} in thread i. Note that a statement at location ℓ in thread tid is executed when the current location of tid changes from ℓ to $\mathsf{succ}(\ell)$.

Given a conflict $c = (\ell_i^{\text{pre}}, \ell_i^{\text{mid}}, \ell_i^{\text{post}}, \ell_j^{\text{cpre}}, \ell_j^{\text{cpost}})$, let $\text{pre}(c) = \ell_i^{\text{pre}}, \text{mid}(c) = \ell_i^{\text{mid}}, \text{post}(c) = \ell_i^{\text{post}}, \text{cpre}(c) = \ell_j^{\text{cpre}}$ and $\text{cpost}(c) = \ell_j^{\text{cpost}}$. Further, let $tid_1(c) = i$ and $tid_2(c) = j$. To prune all interleavings prohibited by the mutex constraints from P'_{abs} we need to consider all conflicts derived from all mutex constraints. We denote this set as $\mathbb{C}$ and let $K = |\mathbb{C}|$.

Example We have an example program and its flow-graph in Fig. 17 (we skip the statement labels in the nodes here). Suppose in some iteration we obtain $\text{mtx}(\texttt{T1}.[\ell_1{:}\ell_2], \texttt{T2}.[\ell_3{:}\ell_6])$. This yields 2 conflicts: c_1 given by $(\ell_3, \ell_4, \ell_5, \ell_1, \ell_2)$ and c_2 given by $(\ell_4, \ell_5, \ell_6, \ell_1, \ell_2)$. On an aside, this example also illustrates the difficulty of lock placement in the actual code. The mutex constraint would naïvely be translated to the lock $\texttt{lock}(\texttt{T1}.[\ell_1 : \ell_2], \texttt{T2}.[\ell_3 : \ell_6])$. This is not a valid lock placement; in executions executing the `else` branch, the lock is never released.

7.2.2 Constructing new P'_{abs}

Initially, let NFA P'_{abs} be given by the tuple $(Q_{\text{old}}, \Sigma \cup \{\epsilon\}, \Delta_{\text{old}}, Q_{\iota,\text{old}}, F_{\text{old}})$, where

(a) Q_{old} is the set of states $\langle \mathscr{V}_o, ctid, (\ell_1, \ldots, \ell_n)\rangle$ of the abstract program $\mathscr{C}_{abs}$ corresponding to $\mathscr{C}$, as well as $\langle\texttt{terminated}\rangle$ and $\langle\texttt{failed}\rangle$,
(b) Σ is the set of abstract observable symbols,
(c) $Q_{\iota,\text{old}}$ is the initial state of $\mathscr{C}_{abs}$,
(d) $F_{\text{old}} = \{\langle\texttt{terminated}\rangle\}$ and
(e) $\Delta_{\text{old}} \subseteq Q_{\text{old}} \times \Sigma \cup \{\epsilon\} \times Q_{\text{old}}$ is the transition relation with $(q, \alpha, q') \in \Delta_{\text{old}}$ iff $q \xrightarrow{\alpha} q'$ according to the abstract preemptive semantics.

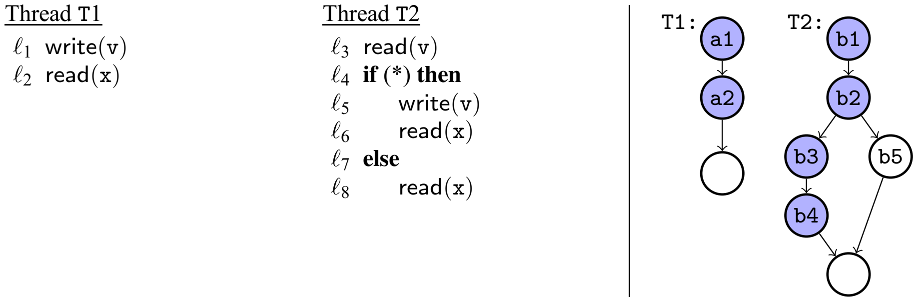

Fig. 17 Example: mutex constraints and conflicts

To enable pruning paths that violate mutex constraints, we augment the state space of P'_{abs} to track the status of conflicts $c_1, \ldots, c_K$ using *four-valued* propositions $p_1, \ldots, p_K$, respectively. Initially all propositions are 0. Proposition p_k is incremented from 0 to 1 when conflict c_k is *activated*, i.e., when control moves from ℓ_i^{pre} to ℓ_i^{mid} along a path. Proposition p_k is incremented from 1 to 2 when conflict c_k *progresses*, i.e., when thread tid_i is at ℓ_i^{mid} and control moves from ℓ_j^{cpre} to ℓ_j^{cpost}. Proposition p_k is incremented from 2 to 3 when conflict c_k *completes*, i.e., when control moves from ℓ_i^{mid} to ℓ_i^{post}. In practice the value 3 is never reached because the state is pruned when the conflict completes. Proposition p_k is reset to 0 when conflict c_k is *aborted*, i.e., when thread tid_i is at ℓ_i^{mid} and either moves to a location different from ℓ_i^{post}, or moves to ℓ_i^{post} before thread tid_j moves from ℓ_j^{cpre} to ℓ_j^{cpost}.

Example In Fig. 17, c_1 is activated when `T2` moves from `b1` to `b2`; c_1 progresses if now `T1` moves from `a1` to `a2` and is aborted if instead `T2` moves from `b2` to `b3`; c_2 completes after progressing if `T2` moves from `b2` to `b3` and is aborted if instead `T2` moves from `b2` to `b5`.

Formally, the new P'_{abs} is given by the tuple $(Q_{\text{new}}, \Sigma \cup \{\epsilon\}, \Delta_{\text{new}}, Q_{\iota,new}, F_{\text{new}})$, where:

(a) $Q_{\text{new}} = Q_{\text{old}} \times \{0, 1, 2\}^K$,
(b) Σ is the set of abstract observable symbols as before,
(c) $Q_{\iota,\text{new}} = (Q_{\iota,\text{old}}, (0, \ldots, 0))$,
(d) $F_{\text{new}} = \{(Q, (p_1, \ldots, p_K)) : Q \in F_{\text{old}} \wedge p_1, \ldots, p_K \in \{0, 1, 2\}\}$ and
(e) Δ_{new} is constructed as follows:
add $((Q, (p_1, \ldots, p_K)), \alpha, (Q', (p'_1, \ldots, p'_K)))$ to Δ_{new} iff
$(Q, \alpha, Q') \in \Delta_{\text{old}}$ and for each $k \in [1, K]$, the following hold:

1. *Conflict activation:* (the statement at location $\text{pre}(c_k)$ in thread $tid_1(c_k)$ is executed)
   if $p_k = 0$, $ctid = ctid' = tid_1(c_k)$, $\ell_{ctid} = \text{pre}(c_k)$ and $\ell'_{ctid} = \text{mid}(c_k)$, then $p'_k = 1$ else $p'_k = 0$,
2. *Conflict progress:* (thread $tid_1(c_k)$ is interrupted by $tid_2(c_k)$ and the conflicting statement at location $\text{cpre}(c_k)$ is executed)
   else if $p_k = 1$, $ctid = ctid' = tid_2(c_k)$, $\ell_{tid_1(c_k)} = \ell'_{tid_1(c_k)} = \text{mid}(c_k)$, $\ell_{ctid} = \text{cpre}(c_k)$ and $\ell'_{ctid} = \text{cpost}(c_k)$, then $p'_k = 2$,
3. *Conflict completion and state pruning:* (the statement at location $\text{mid}(c_k)$ in thread $tid_1(c_k)$ is executed and that completes the conflict)
   else if $p_k = 2$, $ctid = ctid' = tid_1(c_k)$, $\ell_{ctid} = \text{mid}(c_k)$ and $\ell'_{ctid} = \text{post}(c_k)$, then delete state $(Q', (p'_1, \ldots, p'_K))$,

4. *Conflict abortion 1:* ($tid_1(c_k)$ executes alternate statement)
   else if $p_k = 1$ or 2, $ctid = ctid' = tid_1(c_k)$, $\ell_{ctid} = \text{mid}(c_k)$ and $\ell'_{ctid} \neq \text{post}(c_k)$, then $p'_k = 0$,
5. *Conflict abortion 2:* ($tid_1(c_k)$ executes statement at location $\text{mid}(c_k)$ without interruption by $tid_2(c_k)$)
   else if $p_k = 1$, $ctid = ctid' = tid_1(c_k)$, $\ell_{ctid} = \text{mid}(c_k)$ and $\ell'_{ctid} = \text{post}(c_k)$, $\ell_{tid_2(c_k)} = \ell'_{tid_2(c_k)} = \text{cpre}(c_k)$, then $p'_k = 0$

In our implementation, the new P'_{abs} is constructed on-the-fly. Moreover, we do not maintain the entire set of propositions $p_1, \ldots, p_K$ in each state of P'_{abs}. A proposition p_i is added to the list of tracked propositions only after conflict c_i is activated. Once conflict c_i is aborted, p_i is dropped from the list of tracked propositions.

Theorem 7 *We are given a program $\mathscr{C}_{abs}$ and a sequence of observable symbols ω that is a counterexample to preemption-safety, formally $\omega \in \mathscr{L}(\mathsf{P}'_{abs}) \wedge \omega \notin \text{Clo}_I(\mathscr{L}(\mathsf{NP}_{abs}))$. If a pattern P eliminating ω is found, then, after enforcing all resulting mutex constraints in P'_{abs}, the counterexample ω is no longer accepted by P'_{abs}, formally $\omega \notin \mathscr{L}(\mathsf{P}'_{abs})$.*

Proof The pattern P eliminating ω represents a mutex constraint $\text{mtx}(tid_i.[\ell_1{:}\ell''_1], tid_j.[\ell_2{:}\ell''_2])$, such that the trace ω is no longer possible. Mutex constraints represent conflicts of the form $(\ell_i^{\text{pre}}, \ell_i^{\text{mid}}, \ell_i^{\text{post}}, \ell_j^{\text{cpre}}, \ell_j^{\text{cpost}})$. Each such conflict represents a context switch that is not allowed: $\ell_i^{\text{pre}} \rightarrow \ell_i^{\text{mid}} \rightarrow \ell_j^{\text{cpre}} \rightarrow \ell_j^{\text{cpost}} \rightarrow \ell_i^{\text{mid}} \rightarrow \ell_i^{\text{post}}$. Because P eliminates ω we know that ω has a context switch from $tid_i.\ell'_1$ to $tid_j.\ell'_2$, where $\ell_1 \leq \ell'_1 \leq \ell''_1$ and $\ell_2 \leq \ell'_2 \leq \ell''_2$. One of the conflicts representing the mutex constraint is $(\ell_i^{\text{pre}}, \ell_i^{\text{mid}}, \ell_i^{\text{post}}, \ell_j^{\text{cpre}}, \ell_j^{\text{cpost}})$, where $\ell_i^{\text{mid}} = \ell'_1$ and ℓ_i^{pre} and ℓ_i^{post} are the locations immediately before and after ℓ'_1. Further, $\ell_j^{\text{cpre}} = \ell'_2$ and ℓ_j^{cpost} the location immediately following ℓ'_2. If now a context switch happens at location ℓ'_1 switching to location ℓ'_2, this triggers the conflict and this trace will be discarded in P'_{abs}. □

8 Global lock placement constraints

Our synthesis loop will keep collecting and enforcing conflicts P'_{abs} until the language inclusion check holds. At that point we have collected a set of conflicts $\mathbb{C}_{\text{all}}$ that need to be enforced in the original program source code. To avoid deadlocks, the lock placement has to conform to a number of constraints.

We encode the global lock placement constraints for ensuring correctness as an SMT[3] formula LkCons. Let L denote the set of all location and Lk denote the set of all locks available for synthesis. We use scalars $\ell, \ell', \ell_1, \ldots$ of type L to denote locations and scalars $LkVar, LkVar', LkVar_1, \ldots$ of type Lk to denote locks. The number of locks is finite and there is a fixed locking order. Let $\mathsf{Pre}(\ell)$ denote the set of all immediate predecessors in node $\ell : \mathsf{stmt}(\ell)$ in the flow-graph of the original concrete concurrent program $\mathscr{C}$. We use the following Boolean variables in the encoding.

[3] The encoding of the global lock placement constraints is essentially a SAT formula. We present and use this as an SMT formula to enable combining the encoding with objective functions for optimization (see Sect. 9).

$\mathsf{LockBefore}(\ell, \mathit{LkVar})$	$\mathsf{lock}(\mathit{LkVar})$ is placed just before the statement represented by ℓ
$\mathsf{LockAfter}(\ell, \mathit{LkVar})$	$\mathsf{lock}(\mathit{LkVar})$ is placed just after the statement represented by ℓ
$\mathsf{UnlockBefore}(\ell, \mathit{LkVar})$	$\mathsf{unlock}(\mathit{LkVar})$ is placed just before the statement represented by ℓ
$\mathsf{UnlockAfter}(\ell, \mathit{LkVar})$	$\mathsf{unlock}(\mathit{LkVar})$ is placed just after the statement represented by ℓ

For every location ℓ in the source code we allow a lock to be placed either immediately before or after it. If a lock *LkVar* is placed before ℓ, than ℓ is protected by *LkVar*. If *LkVar* is placed after ℓ, than ℓ is not protected by *LkVar*, but the successor instructions are. Both options are needed, e.g. to lock before the first statement of a thread and to unlock after the last statement of a thread. We define three additional Boolean variables:

(D1) $\mathsf{InLock}(\ell, \mathit{LkVar})$: If location ℓ has no predecessor than it is protected by *LkVar* if there is a lock statement before ℓ.

$$\mathsf{InLock}(\ell, \mathit{LkVar}) = \mathsf{LockBefore}(\ell, \mathit{LkVar})$$

If there exists a predecessor ℓ' to ℓ than ℓ is protected by *LkVar* if either there is a lock statement before ℓ or if ℓ' is protected by *LkVar* and there is no unlock in between.

$$\begin{aligned}\mathsf{InLock}(\ell, \mathit{LkVar}) = {} & \mathsf{LockBefore}(\ell, \mathit{LkVar}) \\ & \vee\ (\neg \mathsf{UnlockBefore}(\ell, \mathit{LkVar}) \wedge \mathsf{InLockEnd}(\ell', \mathit{LkVar}))\end{aligned}$$

Note that either all predecessors are protected by a lock or none. We enforce this in Rule C (*C7*) below.

(D2) $\mathsf{InLockEnd}(\ell, \mathit{LkVar})$: The successors of ℓ are protected by *LkVar* if either location ℓ is protected by *LkVar* or $\mathsf{lock}(\mathit{LkVar})$ is placed after ℓ.

$$(\mathsf{InLock}(\ell, \mathit{LkVar}) \wedge \neg \mathsf{UnlockAfter}(\ell, \mathit{LkVar})) \vee \mathsf{LockAfter}(\ell, \mathit{LkVar})$$

(D3) $\mathsf{Order}(\mathit{LkVar}, \mathit{LkVar}')$: We give a fixed lock order that is transitive, asymmetric, and irreflexive. $\mathsf{Order}(\mathit{LkVar}, \mathit{LkVar}') = \mathtt{true}$ iff *LkVar* needs to be acquired before LkVar'. This means that an instruction $\mathsf{lock}(\mathit{LkVar})$ cannot be place inside the scope of LkVar'.

We describe the constraints and their SMT formulation constituting LkCons below. All constraints are quantified over all $\ell, \ell', \ell_1, \ldots \in \mathsf{L}$ and all $\mathit{LkVar}, \mathit{LkVar}', \mathit{LkVar}_1, \ldots \in \mathsf{Lk}$.

(C1) All locations in the same conflict in $\mathbb{C}_{\text{all}}$ are protected by the same lock.

$$\forall \mathbb{C} \in \mathbb{C}_{\text{all}} :\ \ell, \ell' \in \mathbb{C} \Rightarrow \exists \mathit{LkVar}.\ \mathsf{InLock}(\ell, \mathit{LkVar}) \wedge \mathsf{InLock}(\ell', \mathit{LkVar})$$

(C2) Placing $\mathsf{lock}(\mathit{LkVar})$ immediately before/after $\mathsf{unlock}(\mathit{LkVar})$ is disallowed. Doing so would make (*C1*) unsound, as two adjacent locations could be protected by the same lock and there could still be a context-switch in between because of the immediate unlocking and locking again. If ℓ has a predecessor ℓ' then

$$\begin{aligned}\mathsf{UnlockBefore}(\ell, \mathit{LkVar}) &\Rightarrow (\neg \mathsf{LockAfter}(\ell', \mathit{LkVar})) \\ \mathsf{LockBefore}(\ell, \mathit{LkVar}) &\Rightarrow (\neg \mathsf{UnlockAfter}(\ell', \mathit{LkVar}))\end{aligned}$$

(C3) We enforce the lock order according to Order defined in (*D3*).

$$\mathsf{LockAfter}(\ell, \mathit{LkVar}) \wedge \mathsf{InLock}(\ell, \mathit{LkVar}') \Rightarrow \mathsf{Order}(\mathit{LkVar}', \mathit{LkVar})$$

$$\mathsf{LockBefore}(\ell, \mathit{LkVar}) \wedge \left(\bigvee_{\ell' \in \mathsf{Pre}(x)} \mathsf{InLockEnd}(\ell', \mathit{LkVar}') \right) \Rightarrow \mathsf{Order}(\mathit{LkVar}', \mathit{LkVar})$$

(C4) Existing locks may not be nested inside synthesized locks. They are implicitly ordered before synthesized locks in our lock order.

$$(\mathsf{stmt}(\ell) = \mathsf{lock}(\ldots)) \Rightarrow \neg\mathsf{InLock}(\ell, \mathit{LkVar})$$

(C5) No wait statements may be in the scope of synthesized locks to prevent deadlocks.

$$(\mathsf{stmt}(\ell) = \mathsf{wait}(\ldots)/\mathsf{wait_not}(\ldots)/\mathsf{wait_reset}(\ldots)) \Rightarrow \neg\mathsf{InLock}(\ell, \mathit{LkVar})$$

(C6) Placing both $\mathsf{lock}(\mathit{LkVar})$ and $\mathsf{unlock}(\mathit{LkVar})$ before/after ℓ is disallowed.

$$(\neg\mathsf{LockBefore}(\ell, \mathit{LkVar}) \vee \neg\mathsf{UnlockBefore}(\ell, \mathit{LkVar})) \wedge$$
$$(\neg\mathsf{LockAfter}(\ell, \mathit{LkVar}) \vee \neg\mathsf{UnlockAfter}(\ell, \mathit{LkVar}))$$

(C7) All predecessors must agree on their $\mathsf{InLockEnd}$ status. This ensures that joining branches hold the same set of locks. If ℓ has at least one predecessor then

$$\left(\bigwedge_{\ell' \in \mathsf{Pre}(x)} \mathsf{InLockEnd}(\ell', \mathit{LkVar})\right) \vee \left(\bigwedge_{\ell' \in \mathsf{Pre}(x)} \neg\mathsf{InLockEnd}(\ell', \mathit{LkVar})\right)$$

(C8) $\mathsf{unlock}(\mathit{LkVar})$ can only be placed only after a $\mathsf{lock}(\mathit{LkVar})$.

$$\mathsf{UnlockAfter}(\ell, \mathit{LkVar}) \Rightarrow \mathsf{InLock}(\ell, \mathit{LkVar})$$

If ℓ has a predecessor ℓ' then also

$$\mathsf{UnlockBefore}(\ell, \mathit{LkVar}) \Rightarrow \mathsf{InLockEnd}(\ell', \mathit{LkVar})$$

else if ℓ has no predecessor then

$$\mathsf{UnlockBefore}(\ell, \mathit{LkVar}) = \mathsf{false}$$

(C9) We forbid double locking: A lock may not be acquired if that location is already protected by the lock.

$$\mathsf{LockAfter}(\ell, \mathit{LkVar}) \Rightarrow \neg\mathsf{InLock}(\ell, \mathit{LkVar})$$

If ℓ has a predecessor ℓ' then also

$$LockBefore(\ell, \mathit{LkVar}) \Rightarrow \neg\mathsf{InLockEnd}(\ell, \mathit{LkVar})$$

(C10) The end state last_i of thread i is unlocked. This prevents locks from leaking.

$$\forall i : \neg\mathsf{InLock}(\mathsf{last}_i, lk)$$

According to constraints (*C4*) and (*C5*) no locks may be placed around existing wait or lock statements. Since both statements are implicit preemption points, where the non-preemptive semantics may context-switch, it is never necessary to synthesize a lock across an existing lock or wait instruction to ensure preemption-safety.

We have the following result.

Theorem 8 *Let concurrent program $\mathscr{C}'$ be obtained by inserting any lock placement satisfying* LkCons *into concurrent program $\mathscr{C}$. Then $\mathscr{C}'$ is guaranteed to be preemption-safe w.r.t. $\mathscr{C}$ and not to introduce new deadlocks (that were not already present in $\mathscr{C}$).*

Proof To show preemption-safety we need to show that language inclusion holds (Proposition 1). Language inclusion follows directly from constraint (*C1*), which ensures that all mutex constraints are enforced as locks. Further, constraints (*C2*) and (*C6*) ensure that there is never a releasing and immediate reacquiring of locks in between statements. This is crucial because otherwise a context-switch in between two instructions protected by a lock would be possible.

Let as assume towards contradiction that a new deadlocked state $s = \langle \mathscr{V}, ctid, (\ell_1, \ldots, \ell_n) \rangle$ is reachable in $\mathscr{C}'$. By definition this means that none of the rules of the preemptive semantics of $\mathscr{W}$ (Figs. 7, 8) are applicable in s. Remember, that an infinite loop is considered a lifelock. We proceed to enumerate all rules of the preemptive semantics that may block:

- All threads reached their last location, then the TERMINATE rule is the only one that could be applicable. If it is not, then a lock is still locked. This deadlock is prevented by condition (*C10*).
- The rule NSWITCH is not applicable because the other thread is blocked and SEQ is not applicable because none of the rules of the single-thread semantics (Fig. 6) apply. The following sequential rules have preconditions that may prevent them from being applicable.
- Rule LOCK may not proceed if the lock *LkVar* is taken. If $LkVar = ctid$ we have a case of double-locking that is prevented by constraint (*C9*). Otherwise $LkVar = j \neq ctid$. In this case tid_{ctid} is waiting for tid_j. This may be because of
  - (a) a circular dependency of locks. This cannot be a new deadlock because of constraints (*C4*) and (*C3*) enforcing a strict lock order even w.r.t. existing locks.
  - (b) another deadlock in tid_j. This deadlock cannot be new because we can make a recursive argument about the deadlock in tid_j.
- Rule UNLOCK may not proceed if the lock is not owned by the executing thread. In this case we either have a case of double-unlock (prevented by constraint (*C8*)) or a lock is unlocked that is not held by tid_{ctid} at that point. The latter may happen because the lock was not taken on all control flow paths leading to ℓ_{ctid}. This is prevented by constraints (*C7*) and (*C8*).
- Rules WAIT/WAIT_NOT/WAIT_RESET may not proceed if the condition variable is not in the right state. According to constraint (*C5*) ℓ_{ctid} cannot be protected by a synthesized lock. This means the deadlock is either not new or it is caused by a deadlock in a different thread making it impossible to reach signal(*CondVar*)/reset(*CondVar*). In that case a recursive argument applies.
- The THREAD_END rule is not applicable because all other threads are blocked. This is impossible by the same reasoning as above. □

9 Optimizing lock placement

The global lock placement constraint LkCons constructed in Sect. 8 often has multiple models corresponding to very different lock placements. The desirability of these lock placements varies considerably due to performance considerations. For example a coarse-grained lock placement may be useful when the cost of locking operations is relatively high compared to the cost of executing the critical sections, while a fine-grained lock placement should be used when locking operations are cheap compared to the cost of executing the critical sections. Neither of these lock placement strategies is guaranteed to find the optimally performing

program in all scenarios. It is necessary for the programmer to judge when each criterion is to be used.

Here, we present objective functions f to distinguish between different lock placements. Our synthesis procedure combines the function f with the global lock placement constraints LkCons into a single maximum satisfiability modulo theories (MaxSMT) problem and the optimal model corresponds to the f-optimal lock placement. We present objective functions for coarse- and fine-grained locking.

9.1 Objective functions

We say that a statement $\ell : \mathit{stmt}$ in a concurrent program $\mathscr{C}$ is protected by a lock LkVar if $\mathsf{InLock}(\ell, \mathit{LkVar})$ is true. We define the two objective functions as follows:

1. *Coarsest-grained locking* This objective function prefers a program $\mathscr{C}_1$ over $\mathscr{C}_2$ if the number of lock statements in $\mathscr{C}_1$ is fewer than in $\mathscr{C}_2$. Among the programs having the same number of lock statements, the ones with the fewest statements protected by any lock are preferred. Formally, we can define $\mathsf{Coarse}(\mathscr{C}_i)$ to be $\lambda + \epsilon \cdot \mathsf{StmtInLock}(\mathscr{C}_i)$ where λ is the count of `lock` statements in $\mathscr{C}_i$, $\mathsf{StmtInLock}(\mathscr{C}_i)$ is the count of statements in $\mathscr{C}_i$ that are protected by any lock and ϵ is given by $\frac{1}{2k}$ where k is the total number of statements in $\mathscr{C}_i$.
   The reasoning behind this formula is that the total cost is always dominated by the number of `lock` statements. So if all statements are protected by a lock this fact contributes $\frac{1}{2}$ to the total cost.
2. *Finest-grained locking* This objective function prefers a program $\mathscr{C}_1$ over $\mathscr{C}_2$ if $\mathscr{C}_1$ allows more concurrency than $\mathscr{C}_2$. Concurrency of a program is measured by the number of pairs of statements from different threads that cannot be executed together. Formally, we define $\mathsf{Fine}(\mathscr{C}_i)$ to be the total number of pairs of statements $\ell_1 : \mathit{stmt}_1$ and $\ell_2 : \mathit{stmt}_2$ from different threads that cannot be executed at the same time, i.e., are protected by the same lock.

9.2 Optimization procedure

The main idea behind the optimization procedure for the above objective functions is to build an instance of the MaxSMT problem using the global lock placement constraint LkCons such that (a) every model of LkCons is a model for the MaxSMT problem and the other way round; and (b) the cost of each model for the MaxSMT problem is the cost of the corresponding locking scheme according to the chosen objective function. The optimal lock placement is then computed by solving the MaxSMT problem.

A MaxSMT problem instance is given by $\langle \Phi, \langle (\Psi_1, w_1), \ldots \rangle \rangle$ where Φ and each Ψ_i are SMT formulas and each w_i is a real number. The formula Φ is called the *hard constraint*, and each Ψ_i is called a *soft constraint* with *associated weight* w_i. Given an assignment V of variables occurring in the constraints, its cost c is defined as the sum of the weights of soft constraints that are falsified by $V : c = \sum_{i:V \not\models \Psi_i} w_i$. The objective of the MaxSMT problem is to find a model that satisfies Φ with minimal cost. Intuitively, by minimizing the cost we maximize the sum of the weights of the satisfied soft constraints.

In the following, we write $\mathsf{InLock}(\ell)$ as a short-hand for $\bigvee_{\mathit{LkVar}} \mathsf{InLock}(\ell, \mathit{LkVar})$, and similarly $\mathsf{LockBefore}(\ell)$ and $\mathsf{LockAfter}(\ell)$. For each of our two objective functions, the hard constraint for the MaxSMT problem is LkCons and the soft constraints and associated weights are as specified below:

- For the coarsest-grained locking objective function, the soft constraints are of three types: (a) $\neg\mathsf{LockBefore}(\ell)$ with weight 1, (b) $\neg\mathsf{LockAfter}(\ell)$ with weight 1, and (c) $\neg\mathsf{InLock}(\ell)$ with weight ϵ, where ϵ is as defined above.
- For the finest-grained locking objective function, the soft constraints are given by $\bigwedge_{lk} \neg\mathsf{InLock}(\ell, lk) \vee \neg\mathsf{InLock}(\ell', lk)$, for each pair of statements ℓ and ℓ' from different threads. The weight of each soft constraint is 1.

Theorem 9 *For the coarsest-grained and finest-grained objective functions, the cost of the optimal program is equal to the cost of the model for the corresponding MaxSMT problem obtained as described above.*

10 Implementation and evaluation

In order to evaluate our synthesis procedure, we implemented it in a tool called Liss, comprised of 5400 lines of C++ code. Liss uses Clang/LLVM 3.6 to parse C code and insert locks into the code. By using Clang's rewriter, Liss is able to maintain the original formatting of the source code. As a MaxSMT solver, we use Z3 version 4.4.1 (unstable branch). Liss is available as open-source software along with benchmarks.[4] The language inclusion algorithm is available separately as a library called Limi.[5] 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 NP_{abs} and P'_{abs}, which we construct on the fly.

Our prototype implementation has some limitations. First, Liss uses function inlining during the analysis phase and therefore cannot handle recursive programs. During lock placement, however, functions are taken into consideration and it is ensured that a function does not "leak" locks. 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. This is not a limitation of our technique, which could be combined with known aliasing analysis techniques.

We evaluate our synthesis method w.r.t. the following criteria: (1) Effectiveness of synthesis from implicit specifications; (2) Efficiency of the proposed synthesis procedure; (3) Effectiveness of the proposed coarse abstraction scheme; (4) Quality of the locks placed.

10.1 Benchmarks

We ran Liss on a number of benchmarks, summarized in Table 1. For each benchmark we report the complexity [lines of code (LOC), number of threads (Th)], the number of iterations (It) of the language inclusion check (Fig. 12) and the maximum bound k (MB) that was used in any iteration of the language inclusion check. Further we report the total time (TT) taken by the language inclusion check loop and the time for solving the MaxSMT problem for the two objective functions (Coarse, Fine). Finally, we report the maximum resident set size (Memory). All measurements were done on an Intel core i5-3320M laptop with 8 GB of RAM under Linux.

[4] https://github.com/thorstent/Liss.

[5] https://github.com/thorstent/Limi.

10.1.1 Implicit versus explicit specification

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 [6]. In addition, we evaluate Liss and ConRepair on several *new* assertion-based benchmarks (Table 1). We report the time ConRepair took in Table 2. We added `yield` statements to the source code of the benchmarks to indicate where a context-switch in the driver would be expected by the developer. This is a very light-weight annotation burden compared to the assertions required by ConRepair.

Table 1 Experiments

Name	LOC	Th	It	MB	TT (s)	Coarse (s)	Fine (s)	Memory (MB)	CR (s)
ConRepair benchmarks									
ex1.c	18	2	1	1	<1	<1	<1	29	<1
ex2.c	23	2	1	1	<1	<1	<1	29	<1
ex3.c	37	2	1	1	<1	<1	<1	29	<1
ex5.c	42	2	4	1	<1	<1	<1	32	<1
lc-rc.c[c]	35	4	0	1	<1	N/A	N/A	15	9
dv1394.c	37	2	2	1	<1	<1	<1	32	17
em28xx.c	20	2	1	1	<1	<1	<1	29	<1
f_acm.c	54	3	6	1	<1	<1	<1	35	1872
i915_irq.c	17	2	1	1	<1	<1	<1	29	2.6
ipath.c	23	2	1	3	<1	<1	<1	29	12
iwl3945.c	26	3	0	1	<1	<1	<1	15	5
md.c	35	2	1	1	<1	<1	<1	30	1.5
myri10ge.c[c]	60	4	0	3	<1	N/A	N/A	16	1.5
usb-serial.bug1.c	357	7	2	1	6.1	<1	<1	267	∞[b]
usb-serial.bug2.c	355	7	2	1	4.5	<1	<1	175	3563
usb-serial.bug3.c	352	7	2	1	2.8	<1	<1	105	∞[b]
usb-serial.bug4.c	351	7	2	1	3.8	<1	<1	130	∞[b]
usb-serial.c[a]	357	7	0	3	31.9	N/A	N/A	792	1200
CPMAC driver benchmark									
cpmac.bug1.c	1275	5	1	2	6	1.6	1.1	156	
cpmac.bug2.c	1275	5	4	10	152.9	63	41.4	1210	
cpmac.bug3.c	1270	5	9	4	11.1	16.2	9.6	521	
cpmac.bug4.c	1276	5	4	7	107.3	10.5	6.5	5392	
cpmac.bug5.c	1275	5	4	4	136.5	11	7.7	3549	
cpmac.c[a]	1276	5	0	1	2.1	N/A	N/A	114	
memcached benchmark									
memcached.c	294	2	104	2	22.8	6.2	2.1	114	

Th threads, *It* iterations, *MB* max bound, *TT* time for language incl. loop, *CR* ConRepair time
[a] Bug-free example
[b] Timeout after 3 h
[c] Race not detected, as it was present under non-preemptive scheduling

Table 2 Lock placement statistics: the number of synthesized lock variables, lock and unlock statements, and the number of abstract statements protected by locks for different objective functions

Name	No objective			Coarse			Fine		
	Locks	locks/ unlocks	Protected instr	Locks	locks/ unlocks	Protected instr	Locks	locks/ unlocks	Protected instr
cpmac.bug1	2	6/6	11	1	3/3	11	1	3/3	9
cpmac.bug2	2	22/23	119	1	4/4	98	1	6/7	95
cpmac.bug3	1	4/4	29	1	2/3	29	1	5/6	28
cpmac.bug4	4	16/16	53	1	4/4	53	1	6/6	26
cpmac.bug5	3	15/15	30	1	4/4	30	1	5/5	30
memcached	2	5/5	26	1	1/1	28	1	2/2	24

The set includes synthetic 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. For LISS 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, deadlock-free 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 bugs that manifested under non-preemptive scheduling; LISS did not detect these bugs. 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.

10.1.1.1 Performance and precision CONREPAIR uses CBMC for verification and counterexample generation. Due to the coarse abstraction we use, both 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 s, whereas it took CONREPAIR 20 min [6].

The MaxSMT lock placement problem is solved in less than 65s for our choice of objective functions. It is clear that without an objective function the lock placement problem is in SAT, and Z3 solves it in less than 1s in each case. The coarse- and fine-grained lock placement are natural choices, we did not attempt other more involved objective functions.

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 guard variable to model the return value.

10.1.2 Simplified real-world benchmarks

In this section we present two additional benchmarks derived from real-world concurrent programs. Both benchmarks were manually preprocessed to eliminate pointer aliasing.

10.1.2.1 CPMAC benchmark This benchmark is based on a complete Linux driver for the TI AR7 CPMAC Ethernet controller. The benchmark was constructed as follows. 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 concurrency bugs that commonly occur in drivers of this type [5]: a data race where two threads could be concurrently modifying the hardware packet queue, leaving it in invalid state; an IRQ race where driver resources were deallocated while its interrupt handler could still be executing, leading to a use-after-free error; an initialization race where the driver's request queue was enabled before the device was fully initialized, and two races between interrupt enable and disable operations, causing the driver to freeze. 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. These results support our choice of data-oblivious abstraction and confirm the conjecture that synchronization patterns in OS code rarely depend on data values. At the same time, the need for manual refinements indicates that achieving fully automatic synthesis requires enhancing our method with automatic abstraction refinement.

10.1.2.2 Memcached benchmark Finally, we evaluate LISS on *memcached*, an in-memory key-value store version 1.4.5 [19]. The core of memcached is a non-reentrant library of store manipulation primitives. This library is wrapped into the `thread.c` module that implements a thread-safe API used by server threads. Each API function performs a sequence of library calls protected with locks. In this case study, we synthesize lock placement for a fragment of the `thread.c` module. In contrast to our other case studies, here we would like to synthesize locking from scratch rather than fix defects in existing lock placement. Furthermore, optimal locking strategy in this benchmark depends on the specific load. We envision that the programmer may synthesize both a coarse-grained and a fine-grained version and at deployment the appropriate version is selected.

10.1.3 Quality of synthesis

Next, we focus on the quality of synthesized solutions for the two real-world benchmarks from our benchmark set. Table 2 compares the implementation synthesized for these benchmarks using each objective functions in terms of (1) the number of locks used in synthesized code, (2) the number of lock and unlock statements generated, and (3) total number of program statements protected by synthesized locks.

We observe that different objective functions produce significantly different results in terms of the size of synthesized critical sections and the number of lock and unlock operations guarding them: the fine-grained objective synthesizes smaller critical sections at the cost of introducing a larger number of lock and unlock operations. Implementations synthesized without an objective function are clearly of lower quality than the optimized versions: they contains large critical sections, protected by unnecessarily many locks. These observations hold for the CPMAC benchmarks, where we start with a program that has most synchronization already in place, as well as the memcached benchmark, where we synthesize synchronization from scratch.

To summarize our experiments, we found that (1) while our coarse abstraction is highly precise in practice, automatic abstraction refinement is required to further reduce manual effort involved in synchronization synthesis; we leave such extension to future work; (2) additional work is required to improve the performance of our method to be able to handle real-world systems without simplification; (3) the objective functions allow specializing synthesis to a particular locking scheme; (4) the time required to solve the MaxSMT problem is small compared to the analysis time.

11 Conclusion

We introduced a technique to synthesize locks using an implicit specification. The implicit specification relieves the programmer of the burden of providing sufficient assertions to specify correctness of the program. Our synthesis is guaranteed not to introduce deadlocks and the lock placement can be optimized using a static optimization function.

In ongoing work [7] we aim to optimize lock placement not merely using syntactic criteria, but by optimizing the actual performance of the program running on a specific system. In this approach we start with a synthesized program that uses coarse locking and then profile the performance on a real system. Using those measurements we adjust the locking to be more fine-grained in those areas where a high contention was measured.

Acknowledgements Open access funding provided by Institute of Science and Technology (IST Austria). This work was published, in part, in Computer Aided Verification (CAV) 2015 [4] 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.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Alglave J, Kroening D, Nimal V, Poetzl D (2014) Don't sit on the fence—a static analysis approach to automatic fence insertion. In: CAV, pp 508–524
2. Bertoni A, Mauri G, Sabadini N (1982) Equivalence and membership problems for regular trace languages. In: Automata, languages and programming. Springer, Heidelberg, pp 61–71
3. Bloem R, Hofferek G, Könighofer B, Könighofer R, Außerlechner S, Spörk R (2014) Synthesis of synchronization using uninterpreted functions. In: FMCAD, pp 35–42
4. Černý P, Clarke EM, Henzinger TA, Radhakrishna A, Ryzhyk L, Samanta R, Tarrach T (2013) From non-preemptive to preemptive scheduling using synchronization synthesis. In: CAV, pp 180–197. https://github.com/thorstent/Liss
5. Černý P, Henzinger T, Radhakrishna A, Ryzhyk L, Tarrach T (2013) Efficient synthesis for concurrency by semantics-preserving transformations. In: CAV, pp 951–967
6. Černý P, Henzinger T, Radhakrishna A, Ryzhyk L, Tarrach T (2014) Regression-free synthesis for concurrency. In: CAV, pp 568–584. https://github.com/thorstent/ConRepair
7. Černý P, Clarke EM, Henzinger TA, Radhakrishna A, Ryzhyk L, Samanta R, Tarrach T (2015) Optimizing solution quality in synchronization synthesis. ArXiv e-prints. ArXiv:1511.07163
8. Cherem S, Chilimbi T, Gulwani S (2008) Inferring locks for atomic sections. In: PLDI, pp 304–315
9. Clarke EM, Emerson EA (1982) Design and synthesis of synchronization skeletons using branching time temporal logic. Springer, Berlin
10. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: TACAS, pp 168–176. http://www.cprover.org/cbmc/
11. De Wulf M, Doyen L, Henzinger TA, Raskin JF (2006) Antichains: a new algorithm for checking universality of finite automata. In: CAV. Springer, Heidelberg, pp 17–30
12. Deshmukh J, Ramalingam G, Ranganath V, Vaswani K (2010) Logical concurrency control from sequential proofs. In: Programming languages and systems. Springer, Heidelberg, pp 226–245
13. Eswaran KP, Gray JN, Lorie RA, Traiger IL (1976) The notions of consistency and predicate locks in a database system. Commun ACM 19(11):624–633
14. Flanagan C, Qadeer S (2003) Types for atomicity. In: ACM SIGPLAN notices, vol 38. ACM, New York, pp 1–12
15. Gupta A, Henzinger T, Radhakrishna A, Samanta R, Tarrach T (2015) Succinct representation of concurrent trace sets. In: POPL15, pp 433–444
16. Herlihy MP, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Progr Lang Syst (TOPLAS) 12(3):463–492
17. Jin G, Zhang W, Deng D, Liblit B, Lu S (2012) Automated concurrency-bug fixing. In: OSDI, pp 221–236
18. Khoshnood S, Kusano M, Wang C (2015) ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In: International symposium on software testing and analysis
19. Memcached distributed memory object caching system. http://memcached.org. Accessed 01 Jul 2015
20. Papadimitriou C (1986) The theory of database concurrency control. Computer Science Press, Rockville
21. Ryzhyk L, Chubb P, Kuz I, Heiser G (2009) Dingo: Taming device drivers. In: Eurosys
22. Sadowski C, Yi J (2010) User evaluation of correctness conditions: a case study of cooperability. In: PLATEAU, pp 2:1–2:6
23. Solar-Lezama A, Jones C, Bodík R (2008) Sketching concurrent data structures. In: PLDI, pp 136–148
24. Vechev M, Yahav E, Yorsh G (2010) Abstraction-guided synthesis of synchronization. In: POPL, pp 327–338
25. Vechev MT, Yahav E, Raman R, Sarkar V (2010) Automatic verification of determinism for structured parallel programs. In: SAS, pp 455–471
26. Yi J, Flanagan C (2010) Effects for cooperable and serializable threads. In: Proceedings of the 5th ACM SIGPLAN workshop on types in language design and implementation. ACM, New York, pp 3–14